A Methodology for Efficient Multiprocessor System-on-Chip Software Development

Von der Fakultät für Elektrotechnik und Informationstechnik der Rheinisch–Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von
M.Sc.
Jianjiang Ceng
aus Zhejiang/China

Berichter: Univ.-Prof. Dr. rer. nat. Rainer Leupers
Univ.-Prof. Dr.-Ing. Gerd Ascheid

Tag der mündlichen Prüfung: 28.04.2011
Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.
# Contents

## 1 Introduction
1.1 Background .................................................. 1
1.2 MPSoC Software Development Challenge ....................... 3
  1.2.1 Programming Model ...................................... 4
  1.2.2 Application Parallelization ............................ 6
  1.2.3 Mapping and Scheduling ................................ 6
  1.2.4 Code Generation ........................................ 7
  1.2.5 Architecture Modeling and Simulation .................. 7
  1.2.6 Summary ................................................ 8
1.3 Thesis Organization .......................................... 8

## 2 Related Work
2.1 General Purpose Multi-Processor .............................. 11
  2.1.1 Cell ...................................................... 12
2.2 General Purpose Graphics Processing Unit .................... 13
2.3 Embedded Multi-Processor System-on-Chip ...................... 14
  2.3.1 MPCore .................................................. 14
  2.3.2 IXP ...................................................... 14
  2.3.3 OMAP .................................................... 15
  2.3.4 CoMPSoC ............................................... 17
  2.3.5 SHAPES ................................................ 19
  2.3.6 Platform 2012 .......................................... 22
  2.3.7 HOPES .................................................. 23
  2.3.8 Daedalus ............................................... 26
  2.3.9 MPA ..................................................... 29
  2.3.10 TCT ..................................................... 32
2.4 Summary ...................................................... 34

## 3 Methodology Overview
3.1 MPSoC Application Programming Studio (MAPS) ............... 35
  3.1.1 Application Modeling .................................... 36
  3.1.2 Architecture Modeling .................................. 37
  3.1.3 Profiling ............................................... 38
  3.1.4 Control/Data Flow Analysis ............................ 38
  3.1.5 Partitioning ............................................ 39
3.1.6 Scheduling & Mapping ........................................... 40
3.1.7 Code Generation ............................................. 40
3.1.8 High-Level Simulation ....................................... 41
3.1.9 Target MPSoC Platform ...................................... 42
3.1.10 Integrated Development Environment ....................... 42
3.2 Contribution of This Thesis ................................. 42

4 Architecture Model ............................................... 45
4.1 Performance Estimation and Architecture Model ............ 45
4.2 Related Work .................................................. 46
4.3 Processing Element Model .................................... 47
  4.3.1 C Compilation Process .................................... 47
  4.3.2 Operation Cost Table ...................................... 49
  4.3.3 Computational Cost Estimation ........................... 50
4.4 Architecture Description File .................................. 51
  4.4.1 Processing Element Description ......................... 52
  4.4.2 Architectural Description ................................. 52
4.5 Summary .................................................... 54

5 Profiling ........................................................... 55
5.1 Related Work .................................................. 56
5.2 Profiling Process Overview .................................... 57
5.3 Trace Generation ............................................... 58
  5.3.1 LANCE Frontend ........................................... 58
  5.3.2 Instrumentation ............................................ 59
  5.3.3 Profiling Runtime Library ................................ 60
  5.3.4 Trace File ................................................ 61
5.4 Post-Processing ............................................... 62
  5.4.1 IR Profile Generation .................................... 63
  5.4.2 Source Profile Generation ............................... 66
  5.4.3 Call Graph Generation .................................... 68
5.5 Summary .................................................... 69

6 Control/Data Flow Analysis .................................. 71
6.1 Control Flow Graph ........................................... 72
6.2 Statement Control Flow Graph ............................... 73
6.3 Statement Control/Data Flow Graph ......................... 74
  6.3.1 Data Dependence Analysis ............................. 75
6.4 Weighted Statement Control/Data Flow Graph ............. 76
  6.4.1 Node Weight Annotation .................................. 77
  6.4.2 Control Edge Annotation ................................ 77
  6.4.3 Data Edge Annotation .................................... 78
6.5 Summary .................................................... 78

7 Partitioning ....................................................... 79
7.1 Related Work ................................................ 79
7.2 Task Granularity ............................................. 81
7.3 Coupled Block ............................................... 83
  7.3.1 CB Definition ............................................. 83
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.2 Schedulability Constraint</td>
<td>84</td>
</tr>
<tr>
<td>7.3.3 Data Locality Constraint</td>
<td>84</td>
</tr>
<tr>
<td>7.4 CB Generation and WSCDFG Partitioning</td>
<td>86</td>
</tr>
<tr>
<td>7.4.1 WSCDFG Partition</td>
<td>86</td>
</tr>
<tr>
<td>7.4.2 Partition Optimality</td>
<td>87</td>
</tr>
<tr>
<td>7.4.3 CAHC Algorithm</td>
<td>89</td>
</tr>
<tr>
<td>7.5 Global Analysis</td>
<td>95</td>
</tr>
<tr>
<td>7.6 User Refinement</td>
<td>96</td>
</tr>
<tr>
<td>7.7 Summary</td>
<td>97</td>
</tr>
<tr>
<td>8 High-Level Simulation</td>
<td>99</td>
</tr>
<tr>
<td>8.1 Related Work</td>
<td>99</td>
</tr>
<tr>
<td>8.2 MVP Overview</td>
<td>102</td>
</tr>
<tr>
<td>8.3 MVP Simulator</td>
<td>103</td>
</tr>
<tr>
<td>8.3.1 Virtual Processing Element</td>
<td>103</td>
</tr>
<tr>
<td>8.3.2 Generic Shared Memory</td>
<td>106</td>
</tr>
<tr>
<td>8.3.3 User Interface and Virtual Peripheral</td>
<td>107</td>
</tr>
<tr>
<td>8.4 Software Tool Chain</td>
<td>108</td>
</tr>
<tr>
<td>8.4.1 Execution Environment</td>
<td>108</td>
</tr>
<tr>
<td>8.4.2 Code Generation</td>
<td>109</td>
</tr>
<tr>
<td>8.5 Debug Method</td>
<td>113</td>
</tr>
<tr>
<td>8.6 Summary</td>
<td>114</td>
</tr>
<tr>
<td>9 Case Study</td>
<td>115</td>
</tr>
<tr>
<td>9.1 JPEG Encoder</td>
<td>115</td>
</tr>
<tr>
<td>9.1.1 TCT Tools</td>
<td>115</td>
</tr>
<tr>
<td>9.1.2 Profiling</td>
<td>116</td>
</tr>
<tr>
<td>9.1.3 Partitioning</td>
<td>116</td>
</tr>
<tr>
<td>9.1.4 Simulation Result and Further Refinement</td>
<td>118</td>
</tr>
<tr>
<td>9.2 H.264 Baseline Decoder</td>
<td>119</td>
</tr>
<tr>
<td>9.2.1 Target Platform</td>
<td>119</td>
</tr>
<tr>
<td>9.2.2 Profiling</td>
<td>121</td>
</tr>
<tr>
<td>9.2.3 Partitioning</td>
<td>122</td>
</tr>
<tr>
<td>9.2.4 Target Code Generation</td>
<td>122</td>
</tr>
<tr>
<td>9.2.5 Simulation Results</td>
<td>124</td>
</tr>
<tr>
<td>9.3 Summary</td>
<td>126</td>
</tr>
<tr>
<td>10 Summary and Outlook</td>
<td>127</td>
</tr>
<tr>
<td>10.1 Summary</td>
<td>127</td>
</tr>
<tr>
<td>10.2 Outlook</td>
<td>128</td>
</tr>
<tr>
<td>A MAPS Architecture Description XML Schema</td>
<td>131</td>
</tr>
<tr>
<td>A.1 Component Library</td>
<td>131</td>
</tr>
<tr>
<td>A.2 Architecture Model</td>
<td>134</td>
</tr>
<tr>
<td>B Pseudo Code for Profiling</td>
<td>137</td>
</tr>
<tr>
<td>B.1 Code Instrumentation</td>
<td>137</td>
</tr>
<tr>
<td>B.2 Function Call Cost Calculation</td>
<td>138</td>
</tr>
<tr>
<td>B.3 IR Statement Execution Cost Calculation</td>
<td>138</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Our lifestyles have been greatly changed by the various new electronic devices, which continuously appear in today’s market. For example, Personal Navigation Assistants (PNAs) help people to find driving direction without the need to read a map; Personal Digital Assistants (PDAs) are used by people to carry a vast amount of information with them just inside a pocket; Portable Multimedia Players (PMPs) allow people to enjoy music and video contents while they travel; and, last but not least, mobile/smart phones have fundamentally changed the way in which people are connected with each other. Behind these different devices, it is the advancement of technologies, especially the rapidly progressing semiconductor technology, that makes the development and production of such devices possible. Figure 1.1 shows a statistic of the transistor count of some Intel processors introduced in the past forty years.

![Figure 1.1: Moore’s Law & Transistor Count of Some Intel Processors](image.png)
It can be seen that the semiconductor industry has been advancing in a spectacular speed as predicted by Moore’s law [1], i.e., the number of transistors that can be fabricated on a single silicon chip doubles every two years. In order to utilize the massive amount of transistors, several years ago designers started integrating multiple processor cores into one chip. One of the reasons that caused this change is power dissipation. Since a processor’s power dissipation increases dramatically when its clock frequency [2] is raised, it would soon be necessary to use water cooling to cool a system down, if cranking up clock frequency is used as the only way to improve performance. By employing multiple processors, the overall computational power is improved without increasing the clock speed; hence, a balance between performance and energy consumption can be achieved. Today, multiprocessing is seen as the only solution for building high-performance computing systems.

In the embedded area, the trend of multiprocessing appears as Multiprocessor System-on-Chip (MPSoC) [3]. Industrial companies have already developed many MPSoC platforms for multimedia and wireless communication applications. For example, TI’s OMAP (Open Multimedia Application Platform) [4] product line targets the mobile phone and personal multimedia device market. Since its first announcement in 2003, three generations of OMAP processors have been put in the market and used by different mobile phone manufacturers. Apart from that, other companies have also developed their own MPSoC platforms in the past few years, e.g., the Freescale i.MX [5], the ST-Ericsson Nomadik [6], the NXP Nexperia [7], the ATMEL DIOPSIS [8], the SAMSUNG mobile SoC [9] and the Qualcomm Snapdragon [10]. Until now, MPSoC has become the de facto architecture for the design of high performance embedded processors with low energy consumption.

To give an example of how an MPSoC could look like, Figure 1.2(a) shows the block diagram of the upcoming next generation OMAP processor, OMAP44x, which is designed specifically for mobile multimedia telecommunication devices. It mainly consists of two ARM Cortex™-A9 MPCore™-RISC processors, a C64x VLIW DSP based programmable video accelerator, a PowerVR™2D/3D graphics accelerator, an image signal processing processor, a number of peripherals for video, camera, IO, etc., and the interconnection between the functional components. On the right-hand side, Figure 1.2(b) is the block diagram of the latest AMD Phenom™-X4 quad-core processor [11], which contains four identical processor cores, a large block of L3 cache, a memory controller and several interconnection components. Comparing both processors from the architectural perspective, it can be seen that the general purpose multiprocessor has less functionalities built in than that of the embedded MPSoC, although it is constructed with several hundred millions of transistors, which is several times more than the latter is.

There are reasons for MPSoC designers to integrate so many heterogeneous computation and IO components into one chip. Hardware accelerators and programmable DSPs used in MPSoCs are normally optimized for a specific application or an application domain; therefore, they can achieve high performance with low energy consumption, which is crucial for today’s battery driven mobile devices. In addition, the high level integration of the processing elements and IO peripherals greatly reduces the number of the external components which are required to build a complete system; the overall manufacturing cost of the product is thereby reduced, which is very important in the highly cost-sensitive consumer electronics market. Moreover, MPSoCs are not always designed from scratch. Often, only by reusing components from a previous generation or a third party IP vendor, MPSoC designers are able to meet the time-to-market requirement. It can be seen that, in order to meet these stringent design criteria, designers have to develop embedded MPSoCs very differently from general purpose multiprocessors.

Nonetheless, even though the use of multiple application/domain specific processors helps designers
to achieve their design goal, it also brings a significant side effect to them, i.e., software complexity. Presently, the problem of how to do software programming efficiently is deemed to be one of the biggest challenges for MPSoC design [12].

### 1.2 MPSoC Software Development Challenge

For decades, software developers have been trained to think in terms of sequential programming languages and write applications to run in uniprocessor machines. In fact, this sequential method has been proven to be very successful in providing a comfortable and efficient programming environment for software developers. The success is mainly due to the fact that the sequential way of programming is very similar to the natural way of human thinking, and decades of research of compilation techniques has made compilers for high-level programming languages like C/C++ so sophisticated that most of the hardware details of uniprocessor machines can be hidden from programmers. Moreover, graphical Integrated Development Environments (IDEs) like *Visual Studio* [13] or *Eclipse* [14] are nowadays commonly used by programmers, which provide a completely integrated solution from source code editing, compilation to debugging. This further helps to reduce the difficulty of software development. All these factors have contributed to the success of the existing hardware-software ecosystem in the uniprocessor era.

Nevertheless, when MPSoCs are massively employed in new designs, software developers discover that the old programming method cannot keep up with the changes of the platform. The underlying hardware architecture of MPSoC is dramatically different from traditional uniprocessor machines:

- **Multiprocessing**: as the name suggests, MPSoCs have more than one programmable processing elements (PEs) available, whose computational performance needs to be explored by software developers. However, uniprocessor oriented tools can only release a fraction of the potential that MPSoCs have.
• **Heterogeneity:** often, the PEs built in MPSoCs are optimized for one application or an application domain by using special instruction set architectures (ISAs) like VLIW or SIMD. Consequently, due to the huge differences between ISAs, different optimization techniques are required in order to produce efficient code. Unfortunately, for this, compilers designed for uniprocessor machines can barely provide satisfactory support.

• **Memory Architecture:** when programming single-processor machines, software developers seldom need to take care of memories, because they are available in one address space. However, the situation has completely changed in MPSoCs, where memories are organized non-uniformly. Each PE can have its own private locations for storing data which are only needed by itself, and PEs can share some memory blocks for communication purposes. Sometimes, even implementing a simple functionality requires the data to be carefully placed in appropriate locations.

Because of these hardware changes, the traditional holistic compiler solution is not able to support MPSoC programming efficiently. Instead of one compiler for all applications, programmers are faced with a number of tools for just one MPSoC application. They are given a programming environment, which has little in common with a uniprocessor workstation. A number of new challenges need to be solved for the programming of MPSoCs.

### 1.2.1 Programming Model

For any software development process, there is always a description of the target application, which is created by using a given programming model. Programming models define the fundamental elements (such as objects, functions, variables, etc.) of a program and the steps that can be used to compose a computation (such as assignment, branches, etc.). Developers use programming models to create application descriptions which are translated by software tools like compiler into machine executable binary code. Eventually, how fast the binary executable can run, i.e., the application performance, strongly depends on the origin of the whole development process, programming models. Generally, they can be classified into two categories, sequential and parallel models.

#### Sequential Model

In the past few decades, the dominant programming model has always been the sequential model. For uniprocessor machines, it just fits naturally, because the hardware beneath can do only one thing at a time. No matter how complex an application is, it must be programmed in a way that everything is done one after another. As a matter of fact, nearly 90% of the software developers nowadays use the sequential C and C++ [15] programming languages. Compiling an application written in such sequential programming languages for a single processor is a mature field. Even for processors with complex micro-architectures, much progress has already been made in the past few years. Different optimization techniques have been proposed for superscalar processors [16], for DSPs [17], for VLIW processors [18] and for exploiting SIMD instructions [19]. Due to the advanced compiler technologies, the hardware details of a single processor are well hidden from programmers when a sequential programming model is used.

The maturity of single processor compilation techniques has made programmers accustomed to sequential programming. For decades, they have been educated to program sequentially for unipro-
It is not easy for them to switch from the way they used to work and describe applications in a parallel manner. Apart from that, there are millions of lines of sequential legacy code that will not be easily rewritten within a short period of time. All these indicate that the sequential programming model will continue to be widely used for a long period of time.

In the MPSoC area, the underlying architectures of multiprocessor machines are so different from that of the uniprocessor ones that the old method cannot hide the hardware details any more and shows its limitation. When an application is written in programming languages like C, its intrinsic parallelism is completely hidden due to the sequential control-flow oriented semantics. Compilers have a hard time parallelizing such a sequential program, not only because of the hidden parallelism but also because of the lack of the application knowledge, which makes it difficult to find the most suitable parallelization strategy. In the meantime, parallel programming models have been used in the high-performance computing (HPC) area, where parallel processing is the only way to make use of those massively parallel machines. For developers, it is natural to think of using a parallel programming model, which better matches the underlying hardware.

**Parallel model**

Historically, parallel programming has been considered to be “the high end of computing” and has been used to solve difficult scientific and engineering problems like atmosphere, physics, bioscience, etc. Such problems often share one character in common, inherent parallelism, i.e., a big problem can be divided into a number of small sub-problems to be solved simultaneously. To model them, several parallel programming models are commonly used in the HPC community, which are threads (e.g., POSIX Threads - PThreads [20]), message passing (e.g., Message Passing Interface - MPI [21]) and data parallel (e.g., Open Multi-Processing - OpenMP [22]).

Nevertheless, the acceptance of these models in the embedded community is still limited. The main obstacle is the overhead of their implementation. Unlike scientific problems, which usually feature massive parallelism, embedded applications do not often show the regularity that can be easily exploited for parallelization. Even for those streaming data processing oriented embedded multimedia applications, their granularity is often too fine to afford the overhead introduced by complex software stacks for implementing a sophisticated parallel programming model like OpenMP. In other words, the time required for communication and synchronization between the parallel computational tasks can be so long that the application can no longer benefit from parallel processing. Therefore, parallel programming models targeting embedded MPSoCs are still needed. To design such a programming model is not trivial.

From the above discussion, it can be seen that MPSoC software development needs the support of the sequential programming model because there are vast amounts of legacy code and programming experiences, which have been accumulated through decades and need to be reused. Nevertheless, for those applications that have inherent parallelism, parallel programming models are required in order to allow programmers to model them efficiently. Due to the difference between the thinking patterns behind these models, it is challenging for programmers to benefit from them simultaneously. At the same time, for MPSoC programming tool developers, it is a major challenge to create a development environment for programmers that is capable of supporting different models.
1.2.2 Application Parallelization

Parallel programming models provide programmers the means to describe applications to run in parallel. As pointed out earlier, years of programming have left developers a huge amount of legacy code written in sequential programming languages. To reuse them for MPSoC platforms, it is necessary to transform the sequential specification into parallel by using a suitable parallel programming model. This process is known by programmers as parallelization or partitioning, and it extracts the hidden parallelism out of an application. While a traditional compiler tries to exploit instruction level parallelism (ILP), the goal of the application parallelization for MPSoCs is to extract task level parallelism.

Theoretically, programmers can choose to parallelize the code by hand or use tools that are supposed to do the job automatically. However, the lessons from the HPC domain have already shown that manually developing parallel software is a time consuming, complex and error-prone process. In the embedded domain, where product development mostly has stringent time-to-market, programmers typically do not have enough time to do everything manually. Therefore, there is a need for automatic parallelization tools to assist the programmer with converting serial programs into parallel ones.

Unfortunately, parallelization has never been a straightforward process. To discover parallelism in any form automatically, powerful flow and dependence analysis capabilities are required. Computing these dependencies is one of the most complex tasks inside a compiler, either for uniprocessor or for multi-processors systems. For a language like C, the problem of finding all dependencies statically is NP-complete and in some cases undecidable. The main reason for this is the use of pointers [23] and indexes to data structures that can only be resolved at runtime. Without accurate dependence information, tools can only conservatively assume the existence of dependence, which eventually becomes an inhibitor for parallelism exploration. Moreover, during the parallelization process, not only the behavior of the application needs to be considered, but features of the underlying MPSoC architecture should also be taken into account so as to determine the best granularity for the application. This makes the parallelization problem even more challenging. Because of these difficulties, the existing auto parallelization tools are mostly limited in terms of their performance, flexibility and applicability.

The imperfection of auto parallelization techniques and the big effort for manual parallelization put developers in a difficult situation. How to parallelize application efficiently is still a significant problem both to MPSoC programmers and to programming tool developers.

1.2.3 Mapping and Scheduling

Mapping, in the context of MPSoC, mainly refers to the process of assigning parallel tasks to PEs and logical communication links to physical channels. It can be done either statically or dynamically. When the assignment is decided before execution, the mapping is static; otherwise, it is dynamic. Compared to static mapping, dynamic mapping makes the decision based on the availability of resources, and hence results in higher resource utilization, but less predictability.

For MPSoCs, scheduling means finding an appropriate timing sequence for parallel tasks. Similar to the mapping problem, it can also be solved either statically or dynamically. Typically, mapping and scheduling are correlated, since the change of mapping will influence the load of PEs, which, consequently, will further influence the task scheduling in order to maximize the PE usage or meet
specific timing constraints. Moreover, embedded applications often have real time requirements, e.g., the decoding of a video frame must be finished before it is rendered to the display. The existence of real time requirements will add more complexity to the problem. Therefore, finding an efficient scheduling and mapping solution is also a major challenge in MPSoC software development.

1.2.4 Code Generation

In the context of a classical compiler, code generation refers to the back-end, which is typically the last phase of the compilation process and eventually generates assembly instructions for the application. Here, in the MPSoC domain, it means the generation of the code which can be digested by the target PE toolchain. It was mentioned earlier that embedded MPSoCs are often heterogeneous, i.e., different types of processors can be deployed in the same platform. The PEs components can be developed in-house or acquired through third-party IP vendors, in which case software tools like C compilers, linkers, assemblers, etc., are shipped together. Each individual PE alone might not directly support the programming model used for the overall MPSoC programming. Therefore, after parallelization, mapping and scheduling, PE specific code in the form of either high-level programming language or assembly must be generated to be further compiled by the toolchain of the corresponding PE, and the PE compiler can then turn on its own optimization features to further improve the code quality.

During the MPSoC code generation process, PE specific communication/synchronization primitives need to be inserted to the generated code in order to implement the high-level programming model. Because of the heterogeneous nature of the underlying MPSoC, the same high-level feature like semaphore or message queue, might be differently supported by PEs. It is therefore important for the MPSoC code generator to be aware of such differences and generate the target code reliably.

In addition, scheduling is crucial for realtime applications. It can be implemented by the platform in hardware, in software, or in a mixture of both. No matter in which form the scheduling is supported, the generator needs to be platform aware and generates the appropriate code to use the available features. Sometimes it can also happen that no existing scheduling solution is provided by the PE vendor. In such case, the generator needs to synthesize a scheduler for the application, which is even more challenging.

1.2.5 Architecture Modeling and Simulation

Generally speaking, an architecture model is an abstraction of the underlying hardware. According to the use of the model, the abstraction can be done at different levels. For example, a traditional uniprocessor compiler needs a model which can tell it about the pipeline architecture of the target processor, so that the instruction scheduling can be determined; conversely, a model which simulates the behavior of the instruction-set of the target processor is often used by developers to run software without a real hardware. Usually, based on the employed modeling technique, architecture models can be categorized as the following.

- **Abstract model**
  
  Abstract models describe the target platform on a very high level with little or no architecture
details. They are typically used to roughly estimate the execution time of the application, so that the developer can find performance bottleneck or other problems early. Since little hardware detail is included in abstract models, no major effort is required to create them. At the same time, it must be noted that the absence of detail is a double-edge sword. The accuracy of such models is often not very high, and this is particularly problematic in the presence of caches and other non-deterministic architectural features.

- **Simulation model**
  Simulation models are software simulators which are able to mimic the behavior of hardware in the host machine. They typically employ the instruction-set simulation (ISS) technique to run the machine code compiled for the target architecture. Embedded application developers often use them to run applications instead of hardware, because less effort is required to build a software simulator than a hardware prototype. In the industry, they are also called *Virtual Platforms* (VPs). The accuracy of VPs depends on the amount of details they simulate. The more, the better. However, the speed of simulation decreases, when more hardware details are simulated. Besides, a more detailed simulator often requires more time to be developed. Therefore, VP developers have to carefully balance these issues and get the best trade-off.

Because of the limitation of various modeling methods, it is difficult to develop one golden model that can do everything for programmers. In the whole MPSoC software development process, multiple models of the target platform are required. For instance, abstract models can be used early in the design to roughly estimate the effect of different scheduling and mapping scenarios, and detailed simulation models can be used to validate the result predicted by the high-level model. Programmers need to identify the requirements and carefully select the corresponding model. For the developer of MPSoC architecture models, it is a bigger challenge to create models under the conflicting measures such as speed, accuracy, flexibility, usability, etc.

### 1.2.6 Summary

Overall, MPSoC programming is a very complex process that poses a lot of challenges from different aspects of software development. Variations of some of the problems might have been studied in different contexts, e.g., the parallel programming model has been studied in the HPC community for decades. However, due to the huge difference between MPSoCs and traditional computational platforms like general purpose multiprocessor machines, the old solutions are not directly applicable for the new platforms. These challenges must be studied in a pure MPSoC focused context. A systematic methodology is desired in order to provide programmers a satisfactory solution for the efficient development of MPSoC software.

### 1.3 Thesis Organization

Within the range of this thesis, a software development framework, called *MAPS*, is proposed in order to assist programmers in developing applications for MPSoCs. The rest of this thesis will present the details of the methodology and the implementation of the components of MAPS. After the introduction of the MPSoC and its programming challenges discussed in the previous section, Chapter 2 presents a survey on the existing industrial platforms and academic research works, which are related to MAPS. An overview of the proposed methodology is first given in
Chapter 3 before the discussion of each individual component of the framework. Afterwards, the MAPS architecture model, which is used as input to provide high-level architectural information, is introduced in Chapter 4. Chapter 5 discusses the profiling technique that is employed by MAPS to characterize the target application in the form of a profile for both the programmer and MAPS itself. To prepare the parallelization of the application, MAPS performs sophisticated control and data flow analysis, whose details are given in Chapter 6. The parallelization method supported by MAPS is discussed in Chapter 7. In order to test the software early in the design flow, a high-level abstract MPSoC simulator, called MAPS Virtual Platform (MVP), is developed as part of the framework. Chapter 8 presents the details of the MVP. The applicability and the usability of the MAPS framework and methodology have been tested by using several MPSoC platforms case studies, and the results are presented in Chapter 9. Finally, Chapter 10 summarizes the entire thesis and provides an outlook on future work.
Chapter 2

Related Work

As it has been discussed in the previous chapter, MPSoC is today’s de facto standard architecture for embedded applications, and the growth of its complexity causes its software development to become the biggest challenge in its design. Considering that the problem is still deemed to be largely unsolved [24], researchers and engineers have done a lot of work in the past few years in order to find a satisfactory solution. This chapter gives a survey on different approaches, which have been proposed by both academic research groups and industrial companies, and are related to the work done in this thesis. Special attention is paid to some key aspects like the choice of programming model, parallelization method, scheduling and mapping exploration, code generation support, etc.

In the rest of this chapter, Section 2.1 first discusses general purpose multi-processors which are typically employed in desktop like environments. Then, Section 2.2 gives a brief introduction to general purpose Graphics Processing Units. The focus of this chapter is on embedded MPSoCs and their programming tools which are discussed in detail in Section 2.3. Finally, Section 2.4 summarizes the whole chapter.

2.1 General Purpose Multi-Processor

General purpose multi-processors such as Intel Core™2 Quad [25] and AMD Phenom™II X4 [11], are not specifically optimized for one application or application domain. Typically, they are installed in a desktop or workstation environment where the user can use them to run any kind of application.

Normally, the architecture of such general purpose multi-processors are symmetric. For example, Figure 2.1 shows the architecture of an Intel Core™2 Quad processor, in which four identical processor cores are employed. In order to achieve a high performance, they spend a huge amount of transistors to construct sophisticated micro-architecture and integrate large cache memories\(^1\). Strictly speaking, such multi-processors are not SoC, because the extra components such as memory and I/O peripheral, which are required to compose a complete system, are not integrated.

\(^1\)The latest Intel quad-core processor, i7-975, consists of 731 million transistors [26]
The programming of general purpose multi-processors is nowadays mostly still manual. To explore the computational power of multiple processor cores, the programmer needs to directly write code with parallel programming API or languages such as PThread [20], OpenMP [22] and MPI [21]. As a concrete example, the following paragraphs discuss the Cell multi-processor [27] in detail.

2.1.1 Cell

The Cell processor was jointly developed by IBM, Sony, and Toshiba for both consumer and high-performance computing market. It is employed in Sony’s PlayStation 3 game console and some server products of IBM. The architecture of the microprocessor differs from that of a typical Intel multi-processor with respect to processing element and memory. The processing elements used in the Cell processor are heterogeneous, and the memory architecture of the Cell is nonuniform. Figure 2.2 shows the architecture of the processor.

The Cell multi-processor consists of one Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). The PPE is a 64-bit PowerPC processor, and can run 32-bit and 64-bit operating systems and applications. The SPE features a single instruction multiple data architecture which is optimized for running computation-intensive applications. A on-chip interconnection bus connects all the processor elements and allows them to access the shared main memory. Additionally, each SPE has a so-called local storage (LS) as private memory.
The architecture of the Cell processor is tailored to support mainly two levels of parallel execution, task and instruction. IBM provides both open source and proprietary programming tools to Cell software developers to exploit the potential of the processor. At task level, OpenMP is used by programmers to write parallel applications for the Cell processor. The programmer still needs to decide where and how to parallelize the target application. At instruction level, the SPE compiler is able to utilize the SIMD instructions automatically, when the programmer follows the related coding guidelines carefully [28].

2.2 General Purpose Graphics Processing Unit

Graphics Processing Units (GPUs) are originally designed to accelerate pixel processing of 3-Dimensional (3D) graphics. Since the pixels of a 3D scene can naturally be processed in parallel, GPUs often feature a massively parallel architecture. In the past few years, they are more and more used for general purpose computing, which makes them general purpose graphics processing unit (GP-GPU). Nvidia CUDA [29] and ATI Stream [30] are examples of such GP-GPU technology.

The architecture of GP-GPUs is highly parallel. The ATI Radeon™HD5970 graphics card [31], for example, has 1,600 so-called thread processors per GPU. Figure 2.3 shows a simplified diagram of an ATI GPU. It consists of a group of SIMD engines, one SIMD engine contains numerous thread processors, and each thread processor is a five-issue VLIW processor. All thread processors within a SIMD engine execute the same instruction stream. Different SIMD engines can execute different instruction streams, and they are all managed by the thread dispatch processor.

Notice that multiple thread processors must share a single instruction stream. In comparison to traditional multi-processors which are capable of running different programs simultaneously, this is a big difference, and special support are required to efficiently program GP-GPUs. For example, the ATI stream processor requires the GPU executed code, which is called thread kernel, to be wrapped in a function with array or vector parameters. Besides, new languages such as OpenCL [32], which is an extension based on the C++ language, have been proposed to support GP-GPU programming.

![Figure 2.3: ATI Stream Processor](image)

Figure 2.3: ATI Stream Processor
2.3 Embedded Multi-Processor System-on-Chip

While general purpose multi-processors and GPUs mostly feature a homogeneous architecture, the situation is completely different in the embedded domain. Today, battery driven personal wireless communication and entertainment devices have a strong requirement on performance under a stringent energy consumption budget. To tackle the challenging requirement, embedded MPSoCs often employ application specific processing elements and are highly integrated. Besides, the time-to-market window for consumer electronic products is very small, the platform-based MPSoC design methodology is often preferred by companies. That is, the new generation of the MPSoC architecture will be based on the previous successful model with some evolutionary improvements. Until today, many MPSoC platforms have been developed and put into the market by different companies, e.g. ATMEL Diopsis [8], Freescale i.MX51 [33], Infineon XGold SDR [34], NXP nexperia [7], ST Nomadik [6], Qualcomm Snapdragon [10], etc. In order to program these MPSoCs efficiently, a lot of work has been done by companies and research institutes.

2.3.1 MPCore

MPCore is an embedded multi-processor IP provided by ARM, which can be used by different MPSoC vendors in their designs. The Cortex™-A9 MPCore [35] is the latest generation in the product line. Both TI and ST have deployed the multi-processor in their latest MPSoC chips [36], [37]. MPCore is configurable. Depending on the target application, MPSoC architects can choose to integrate up to 4 processor cores into their designs. Figure 2.4 shows the structure of a quad-core MPCore. As it can be seen from the figure, MPCore features a symmetric multi-processing architecture which is similar to the general purpose multi-processors used in desktop computers and eases software development.

From software perspective, MPCore is supported by the Linux OS [38]. On top of this, OpenMP and MPI are available for programmers to write parallel applications. However, ARM as an IP vendor does not provide programming tools specifically designed for MPCore. There are several academic projects, such as HOPES (Section 2.3.7) and MPA (Section 2.3.9), which use MPCore as the target platform for parallel programming researches.

2.3.2 IXP

The Intel IXP processor [39] is a network processor which is specifically targeted at the networking application domain. As a generic packet processor, the IXP has its functions optimized for tasks...
which are typically found in network protocol processing, such as packet queue management, data bitfield manipulation, etc. Figure 2.5 shows the structure of the IXP2400 processor which contains one Intel XScale core and eight so-called Micro-Engines (MEs) arranged in two clusters. The XScale core is a RISC processor with an ARM compatible instruction-set. Its intended use in the IXP is to control and support the processes running on the MEs. Each ME is a simple RISC-processor with a instruction set specially tuned for processing network data. Dedicated communication channels are built between neighboring MEs to accelerate their data exchange.

To program the processor cores in the IXP, two separate compilation tool chains are provided by Intel [40]. Programmers can use C to write code for the XScale and the ME, although, only a subset of C is supported for the ME. Besides, when the ME packet processing instructions are required in the application, they need to be invoked through either special compiler known functions or assembly routines, which is not very convenient. There are research works done to ease the programming of the IXP. For instance, the NP-Click tool [41] uses a special language to describe network applications, which is more intuitive than assembly and compiler known function. The NP-Click authors estimated a four-fold productivity increase using NP-Click, and demonstrated performance between 60% and 99% of designs directly coded with the Intel tools.

2.3.3 OMAP

The TI OMAP (Open Multimedia Application Platform) product family is one of the earliest heterogeneous MPSoCs which are productized. The whole product line targets the mobile multimedia communication device market, and is upgraded regularly. Since the first generation, the OMAP1510 processor, was revealed in 2002, three generations of OMAP products have already been put into the market, and the fourth generation is on its way. If different generations of the OMAP platform are compared with each other, it can been seen that the basic structure of the platform has not been changed that much, which shows a paradigmatic example of platform-based design.

Figure 2.6 shows the high-level architecture of the OMAP1510 and the OMAP3630 processors, which are the first and the third generation of the product line and both consist of a RISC processor from ARM and an DSP from TI itself. Though the newer generation features more peripherals and accelerators for application domains like 3D, image processing and so on, the main programmable elements of the MPSoCs are similar. If the next generation OMAP, the OMAP44x processor whose diagram is shown in Figure 1.2(a), is taken into account together, one can easily see that the idea behind the OMAP platform is basically using a combination of RISC and DSP processors for control-centric and data-centric applications respectively. Thereby, a single chip is able to
support both general purpose and multimedia applications efficiently.

The OMAP platform is designed to be flexible so that device manufactures can differentiate their products from others through the software running on the platform. Therefore, the RISC processor is completely open to developers, who are free to run any available OS and applications on it. However, the choice of the DSP software solution, on the other hand, is very limited. There is only one OS available for the DSP in OMAP, which is the DSP/BIOS developed by TI itself. The communication between the RISC and the DSP is realized through a communication layer called DSP Bridge, which on the DSP side is part of the OS. On the RISC side, a DSP Bridge driver is required to allow software on the RISC processor talking to the DSP. A simplified overview on the OMAP software stack is shown in Figure 2.7.

There are reasons for such a heterogeneous software architecture. DSP programming is known to be more difficult than programming standard RISC processors, when performance is considered. Running a piece of unoptimized code on DSP might not bring any speed-up than on a RISC processor. Expertise and time is required to write code for DSPs. As a result, in order to help customers exploiting the potential of the DSP, TI provides his own software stack. Besides, TI has a large collection of highly optimized DSP software libraries, which is accumulated over years. It would be a loss, if it is not reused. On the one hand, such heterogeneous software stack ensures the maximum reuse of existing DSP routines, on the other hand, it makes parallel programming on OMAP more difficult.

- **Programming Model**
  Strictly speaking, there is no parallel programming model currently supported by OMAP. In order to write an application using both the RISC and the DSP, the programmer needs to separately write code for both processors and use different compiler tool-chains and libraries to build the executable binaries respectively, which is a relatively tedious process.

- **Parallelization**
  Since the low-level software running on the RISC processor of OMAP completely depends on the choice of device manufactures, the final application runtime environment can only be known after the product or at least a complete prototype is finished. It is difficult for developers to write a tool to parallelize applications to run in an unknown environment. No work has been published so far, which is able to support parallelization for OMAP.

![Figure 2.6: Comparison of the 1st and 3rd Generation OMAP Processors](image_url)
2.3. EMBEDDED MULTI-PROCESSOR SYSTEM-ON-CHIP

![OMAP Software Stack Diagram](image)

Figure 2.7: OMAP Software Stack

- **Mapping and Scheduling**
  Task mapping in OMAP is in principle not a problem for tools but for programmers, because softwares for the RISC and the DSP are written separately by hand. Programmers need to use their experiences to decide which processor should do what. Scheduling support on the RISC processor depends on device manufactures, who can choose to implement a sophisticated OS or a lightweight one. Mostly, the choice is dependent on the demands of the market. On the DSP side, the TI DSP/BIOS is a multi-tasking realtime OS, which supports both static and dynamic scheduling. The scheduler of DSP/BIOS is preemptive, tasks can be assigned with different priorities in order to consume different amounts of processing time.

- **Modeling and Simulation**
  OMAP is one of the earliest MPSoCs, which get virtual platform support. Start from the first generation of OMAP, Virtio [42], which is now part of Synopsys, has been providing OMAP virtual platforms to developers. The virtual platform built by Virtio is a very detailed full-system simulator, it models not only the MPSoC itself but also all other components, from external memories to mechanical switches, which are required to build a complete prototype. Because of the use of ISS technology, running unmodified target binaries is supported.

Overall, although TI provides a rich set of tools for the programmer to develop applications using different processing elements and provide many abstractions to hide as much platform details as possible, how to partition and allocate different tasks/applications onto the platform so as to efficiently leverage the underlying processing power still highly relies on the knowledge and experience of the programmer. Programming OMAP processors as a whole is not a trivial job.

2.3.4 CoMPSoC

The CoMPSoC (Composable and Predictable Multi-Processor System on Chip) platform [43] is jointly developed by NXP and the Eindhoven University of Technology. It aims at providing a scalable hardware and software MPSoC template with **composability** and **predictability**. Figure 2.8 shows the basic hardware architecture of the CoMPSoC platform. CoMPSoC has a configurable number of PEs, which are Silicon Hive Very Long Instruction Word (VLIW) processors [44]. The processor core does not support preemptive multitasking, and static scheduling is used to share a PE between tasks. Memories in CoMPSoC are distributed. Each PE has its own local memory, and a SRAM is available in the platform as global shared memory. The interconnection between the hardware modules is realized through the Æthereal NoC [45]. In order to ensure that all applications can get bandwidth on the NoC, CoMPSoC uses time division multiplexing to control the communication between its components.
The software stack used by the CoMPSoC platform is the C-HEAP API [46], which handles the communication between PEs, shared memories and peripherals. It is implemented in a distributed manner for the sake of scalability and synchronization efficiency.

- **Programming Model**
  When programmers write software with the C-HEAP API for CoMPSoC, the underlying programming model of the application is the so-called Kahn Process Network (KPN) [47]. KPN models each application as a set of processes, which communicate exclusively with each other via unidirectional FIFO channels. Originally, the FIFO channels defined in a KPN have an infinite length, which implies that a channel cannot be full. Nonetheless, in order to have an efficient implementation, some restrictions are imposed by the C-HEAP API. For example, the communication FIFOs between processes are bounded, and a write operation on a full FIFO will block the parent process from further execution.

- **Parallelization**
  Parallelizing applications for the CoMPSoC platform is currently still done manually. The developer follows a typical manual parallelization flow consisting of profiling, hot-spot identification and manual code partitioning. No auto-parallelization facility is reported so far.

- **Mapping and Scheduling**
  For the CoMPSoC platform, task mapping and scheduling are done statically before the application is executed. Although the C-HEAP API defines primitives for dynamic task creation and termination, these features are not implemented on CoMPSoC. This mainly due to the restriction of the VLIW PE of the platform, which supports only static scheduling. Besides, sharing one PE between different applications is at the moment not supported in CoMPSoC.

- **Code Generation**
  The code generation of the CoMPSoC platform covers both hardware and software. Based on a high-level system description, HDL code is generated, which implements the hardware of the platform. Since the application for CoMPSoC is written in C, it can be directly compiled to binary. No code needs to be generated for the application itself. Nevertheless, in order to configure the cores and the NoC, a piece of configuration code is generated at design time.
2.3. EMBEDDED MULTI-PROCESSOR SYSTEM-ON-CHIP

- **Modeling and Simulation** For early testing purpose, a transaction-level simulation model of the platform is generated during the code generation process along with the RTL level hardware description. The generated accurate system model is mainly used to test the application software before the Very Large Scale Integration (VLSI) implementation is available. Besides, FPGA emulation is also employ in the design flow of CoMPSoC for verification purposes.

Since *composability* and *predictability* are the main goals of CoMPSoC, the design of the platform inevitably makes some compromises to avoid unpredictability. For example, only static scheduling is used and no PE is shared between different applications. Besides, although the configuration code for CoMPSoC is generated automatically, the application development itself including parallelization is still manual.

### 2.3.5 SHAPES

SHAPES [48] is an European Union FP6 Integrated Project whose objective is to develop a prototype of a massively scalable HW and SW architecture for applications featuring inherent parallelism. The fundamental building blocks of SHAPES are processing element *tiles*, each of which is a self-contained sub-system. Figure 2.9 a) shows the basic configuration of a tile, which is composed of an ATMEL mAgic VLIW floating-point DSP, an ARM9 RISC processor, on-chip memory, and a network interface for on- and off-chip communication. Because of the inclusion of two different types of processors, it is called RISC-DSP Tile (RDT). On the bases of RDT, the RISC-Elementary Tile (RET) and the DSP-Elementary Tile (DET) are developed at the same time, which contain only the RISC or the DSP processor in order to get specialized computational characteristic. Using tiles as components, a multi-tile chip (Figure 2.9 b) can be constructed by putting multiple of them in one chip and using an on-chip network for interconnection. Furthermore, a large number of multi-tile chips can be linked together through a scalable off-chip interconnection network, which eventually forms a massive parallel system (illustrated in Figure 2.9 c) with a very high theoretical aggregated computational performance.

In order to leverage the potential performance of the SHAPES system, a sophisticated software environment is developed by several research groups of different European universities and companies. An overview of the SHAPES software development flow is illustrated in Figure 2.10 ([49]).
The first phase is the Distributed Operation Layer, in short DOL, whose input includes an application specification, an architecture specification and a list of mapping constraints. The DOL is supposed to accomplish several tasks for the SHAPES programmers, namely, functional simulation, mapping and scheduling exploration and performance estimation. Using a multi-objective optimization framework [50], DOL tries to find the best task-to-processor mapping with the consideration of various conflicting criteria such as throughput, delay, predictability, etc. Below DOL is the Hardware-dependent-Software layer (HdS), which provides OS services and realization of communication protocols to DOL. Given the exploration result from DOL, it generates the necessary dedicated communication and synchronization primitives for the underlying SHAPES hardware, configures the operating system for PEs in the SHAPES tiles, and eventually compiles/links everything together into a binary image which can be loaded by either the SHAPES hardware prototype or the Virtual SHAPES Platform (VSP), i.e. an ISS based simulation model of SHAPES. Both the HW prototype and the VSP are capable of running the SHAPES software and providing performance feedback to DOL for the next iteration of the SHAPES SW exploration loop.

- **Programming Model**

DOL also uses the Kahn Process Network [47] as its programming model. The application specification is composed of two parts, an XML file and C/C++ code. The XML file is used to specify the topology of the process network which includes processes, channels and the connections between them. The functionality of processes is specified in the C/C++ source code. A set of coding rules must be fulfilled for the correct specification of process behavior. In each process there must be a `init` and a `fire` function. The `init` function allocates and initializes data, it is called once during the initialization of the application. The `fire` function
is invoked repeatedly afterwards. The communication between processes is realized through
the use of the DOL APIs, e.g. DOL_read() and DOL_write(); as mentioned earlier, these
functions are blocking.

Figure 2.11 a) shows an example process network XML file of a simple application modeled
with DOL. The whole network consists of two processes, producer and consumer, which
are connected by one software fifo channel. The source code of the processes is specified
separately from the XML file, and Figure 2.11 b) shows the definitions of the init and the
fire functions of the producer process. The process network can be visualized as the diagram
which is shown in Figure 2.11 c).

It can be imagined that using a mixed-language model could take advantage of different
languages. For instance, the XML based process network specification can be supported by
tools like [51] to create a graph editing like programming environment. However, until now,
there is no such support available from SHAPES. The strict definition of the process init
and fire function formalizes the behavior of each process. But, it also makes the reuse of
existing source code for DOL harder, because an existing application might not be written
with such behavior in mind.

- **Parallelization**

Until now, parallelizing application for running on the SHAPES platform is still a manual
process. If a sequential application needs to be parallelized for the platform, the programmer
has to manually write a DOL model for it. There has been no auto-parallelization effort
reported so far.

- **Code Generation**

Since different languages are used in the SHAPES application model, a model compiler
is implemented in the DOL layer in order to unify the descriptions into one format for
simulation or generating binary executables. For functional simulation, it generates SystemC
code out of the XML specification of the process network. The process C/C++ source code is
compiled together with the generated code to get a functional simulator. The code generation
procedure does not require sophisticated control and data flow analysis, because all processes
can only access communication channels explicitly through the predefined DOL API.

- **Mapping and Scheduling**

The mapping and scheduling exploration process in DOL includes 2 phases: performance
evaluation and optimization. The performance evaluation uses data from an analytical ar-
chitecture model, which is fast but not fully accurate, and the VSP, which does detailed
simulation and is more accurate yet slow. Combining information from both high- and low-
level, the estimator then predicts the application performance in different mapping/schedul-
ing scenarios. The goal of the DOL optimization is to find the best scenario. To achieve
this, the optimization objectives have to be given by the designer, and then the DOL will
find the solution by using the evolutionary algorithm through the PISA interface[50]. The
result is generated in XML format and passed to the HdS layer.

Both the RISC processor and the DSP are under the management of the HdS, it provides
DOL with OS services and communication primitives. However, the current scheduling sup-
port of the HdS is limited. For the DSP, only static scheduling is available, and the scheduling
on the RISC processor is non-preemptive, which limits the usability of the platform.

- **Modeling and Simulation**
It is mentioned in the previous paragraph that DOL uses both an analytical model and the VSP for performance estimation. The former is an abstract architecture model omitting many details of the underlying architecture. It is specified in XML format and contains three kinds of information: structural elements such as processors/memories, performance data such as bus throughputs/delays, and parameters such as memory sizes/resource sharing policy. The latter is a virtual platform built with the commercial CoWare[52] virtual platform tools. Both the RISC processor and the DSP are simulated in the VSP by ISS’s. Therefore, the VSP is used not only to provide performance feedback to DOL, but also for running and debugging SHAPES software.

Note that, there is one potential issue in the SHAPES software development flow, i.e. the DOL mapping and scheduling exploration requires information from the simulation result of the VSP, whose execution requires the binary executable to be first compiled. Since the generation of the software image needs to first pass the DOL layer, a circular dependence scenario is thereby established. So far, it is not clear how this issue is resolved. What could be possible is that an unoptimized solution is first generated by the framework as the initial solution to initiate the exploration loop.

2.3.6 Platform 2012

Platform 2012 (P2012) is a project initiated by ST Microelectronics [53], whose goal is to develop an MPSoC platform targeting future embedded applications. Similar to SHAPES, the P2012 platform also uses a tiled architecture. The basic building block of the platform is a so-called customizable P2012 tile (see Figure 2.12a.), which is capable of accommodating up to eight processing elements. Both programmable core and hardware blocks can be integrated into the P2012 cluster. As programmable core, the STxP70 processor with a 4 issue VLIW architecture, is used. Withing the customizable cluster, the processing elements are connected with a high-speed interconnection; furthermore, multiple clusters can be linked in form of a 2D mesh with high scalability as shown in Figure 2.12b.

Since the P2012 platform is still relatively new, details about its programming are still not clear.
2.3. EMBEDDED MULTI-PROCESSOR SYSTEM-ON-CHIP

Though, it is claimed that the platform will support OpenCL and provide programming tools optimized for the embedded many-core environment of P2012. Besides, the European Union research project, 2Parma [54], which is started in January 2010, also works on the programming support of the P2012 platform. The research project will take three years to finish, and the outcome of the research is still to be seen.

2.3.7 HOPES

HOPES is a parallel programming framework developed at Seoul National University [55]. Unlike the SHAPES project, in which hardware and software are developed concurrently, HOPES is designed to support existing multiprocessor architectures. In [56], it is reported that HOPES supports the ARM MPCore [35] and the IBM CELL processor [27], and the support for other processor is already planned. Therefore, it can be seen as a retargetable programming framework.

Figure 2.13 gives an overview on the software development flow of HOPES [57]. The center of the whole procedure is an intermediate format which is called Common Intermediate Code (CIC)[58]. The CIC includes descriptions of both the application (task code) and the target parallel architecture. The programmer can either write the CIC task code manually or use a generator to generate the code from a data-flow application model called PeaCE [59]. The architecture description is specified in an XML file. In the CIC layer, applications are modeled as process networks, and the process to processor mapping is currently done manually. The CIC translator translates the CIC program into C code for each target processor core, which includes phases such as generic API translation and scheduler code generation. Finally, the generated target dependent C code is compiled by processor specific compilation tool chains into executables to run on the target platform.

- **Programming Model**

  Strictly speaking, two programming models are supported by the HOPES framework, namely, the CIC model and the PeaCE model. The core of the framework is the CIC model, which is in principle a process network and designed to be the central intermediate representation enabling the retargetable parallel code generation. The process in the traditional process network definition is called task in the CIC model.
Similar to the SHAPES DOL model, the CIC model also separate the process behavior from the structure of the process network. The former is specified in form of task code, which is C based and supports OpenMP pragmas. The latter is in the format of XML, but mixed with the architecture description. Figure 2.14a) shows the application part of the CIC XML file, which instantiates two tasks. The corresponding task code is shown in Figure 2.14b). Each task code consists of three functions: task_init(), task_go(), and task_wrapup(). The task_init() function is called once when the task is invoked to initialize the task. The task_go() function defines the main body of the task and is executed repeatedly in the main scheduling loop. The task_wrapup() method is called before the task is terminated, and its job is to release the used resources. To further explore more fine-grained parallelism within tasks, OpenMP pragmas can be used in these functions.

The default inter-task communication model is a streaming-based channel-like structure, which uses ports to send/receive data between tasks. Generic APIs for channel access and file I/O are used in the task code to keep code-reusability. A special pragma, which can be used to indicate the use of a hardware accelerator for certain functions, is also available. All those constructs will later be translated or implemented by target dependent translators, depending on the mapping.

Although the programmer can manually write CIC models, HOPES supports generating a CIC model from a PeaCE model, when the latter is already available. The PeaCE model is a mixture of three different models of computation. At top level, a task model is used, which specifies the execution condition of each task and communication requirements between tasks; the internal definition of each task is specified with an extended Synchronous-Data-Flow (SDF) model called SPDF [60]; and the control tasks are modeled with a hierarchical concurrent FSM model (fFSM [61]). With the PeaCE model, the application specification can be well formalized. However, the use of several heterogeneous models of computation

![Figure 2.13: HOPES Parallel Programming Flow](image-url)
and languages could make the resulting model very complex.

- **Parallelization**
  Since the PeaCE model contains already a task level model which can be seen as parallelized, the CIC code generation does not have to parallelize the application. Therefore, the HOPES framework does not support parallelizing sequential applications for the programmer.

- **Mapping and Scheduling**
  In the HOPES framework, task mapping and scheduling are decided manually. The programmer specifies not only the task-to-PE mapping, but also the period, the deadline and the priority of tasks. Besides, the overall scheduling policy is also determined in this phase. This information is then stored in the CIC XML files for target code generation.

  Recently, a work has been published by the HOPES research group, which is about the MP-SoC mapping/scheduling techniques for pipelined data-level parallel tasks [62]. It proposes the use of so-called Quantum-inspired Evolutionary Algorithm (QEA) for solving the problem automatically. So far, it is not clear if the algorithm will be employed in the HOPES framework.

- **Code Generation**
  The CIC translator combines the information from the target independent task code and the CIC XML, and generates target dependent C code. The generic communication and I/O APIs are translated into target specific implementations according to the architectural information in the XML file. For the OpenMP pragmas, if the target processor tool-chain provides the corresponding support, the translator will keep the pragmas as they are. Otherwise, the translator will convert them to parallel C code using the services available on the target processor. In case if the target processor does not have an OS, the translator will also synthesize a run-time system to realize task scheduling. All these code will be further compiled by C compilers of the target processors. Although the CIC model is supposed to be architecture independent, the CIC translator is completely target dependent. To support the HOPES flow for a new MPSoC architecture, a new CIC translator needs to be written to realize the translation process, which could be non-trivial.
• **Modeling and Simulation**

For the code generation purpose, a simple architecture model is included in HOPES, and stored in the same CIC XML file as the application process network specification is. It provides information about the address and the size of each memory segment. Besides, memory segments can be assigned to processors as local or shared resources. Except for this, no further detail is available in the specification.

From the above discussion, it can be seen that the HOPES framework mostly focuses on its intermediate representation, the CIC model. Although the model provides the possibility of specifying parallelism, its exploration still relies on the knowledge and the experience of programmers. Besides, the mixture of architecture model and application model in one specification makes CIC models target dependent, which contradicts with its model reusability philosophy, i.e. a CIC model should be reusable for different target MPSoCs.

### 2.3.8 Daedalus

Daedalus [63] is a system-level MPSoC design framework developed at Leiden University. It provides an integrated solution for the hardware/software co-design of MPSoCs. Unlike the previously introduced projects whose target architectures are fixed, the hardware architecture of the target MPSoC is co-designed together with its software in the Daedalus framework. As a result, the output of the framework includes not only C code for the target processors but also RTL code for hardware implementation.

The Daedalus design flow consists of three key components, KPNgen [64], Sesame (Simulation of Embedded System Architectures for Multilevel Exploration) [65] and Espam (Embedded System-level Platform synthesis and Application Modeling) [66]. The relation of these tools are depicted in Figure 2.15. Starting from the application specification, the KPNgen tool helps its user to generate KPN models from sequential C code. The Sesame tool has some restrictions on the input sequential C code, therefore, in case the input C code does not fulfill the requirement, the designer can also directly create the model manually. The created KPN models are stored in XML files, which are first used by the Sesame tool as input.

The goal of the Sesame tool is to do high-level design space exploration for the overall MPSoC design. An IP library is used by it to get information about the available building blocks, and the configuration of the resulting architecture is exported as files in XML format. Along with the hardware architecture specification, the mapping information is also generated by the tool in XML files.

The final HW/SW code generation process is done by the Espam tool with the application KPN, the architecture specification and the mapping specification as input. For hardware implementation, it uses the component RTL models in the library, and generates RTL code for the complete platform, which can be further processed by commercial HW synthesis tools; and for software programming, it produces C code for each target processor, which can be compiled by the corresponding compiler tool-chain. The framework automates the complete HW/SW design process for the MPSoC designer, however, the design space is limited by the component library which currently does not have a large collection of components.

• **Programming Model**
In Daedalus, the application programming model is the KPN model, which is also used in other frameworks. Both the high-level design space exploration and the target HW/SW code generation take KPN models as an input. The framework supports automatically generating KPN models from sequential C code in form of so called static affine nested loop programs (SNLAPs) by the KPNgen tool. A SANLP is a program or a program fragment that is composed of a set of statements, each possibly enclosed in loops and/or guarded by conditions. There are several rules that must be followed to construct a SNLAP. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses can contain enclosing loop iterators and parameters as well as modulo and integer divisions, but no products of these elements, and such expressions are called quasi-affine. Besides, the parameters are symbolic constants, that is, their values may not change during the execution of the program fragment. An example SNLAP is shown in Figure 2.16.

Although the hot spots of scientific, matrix computation, multimedia and adaptive signal processing applications are often SNLAPs, it is seldom that an application is completely written this way. Therefore, for applications outside this class, manual specification is needed.

- **Parallelization**

In Daedalus, the transformation from the sequential C SNLAP code to the KPN model can be seen as a parallelization process. Since the SNLAPs heavily access loops, the KPNgen tool performs array dataflow analysis in order to figure out the loop carried dependences.

The commercial version of the KPNgen tool, the Compaan Hot-spot Parallelizer [67] focuses on the translation of computationally intensive kernels of applications into KPNs. These kernels are also SNLAPs, and the generated KPNs are used for SW/HW implementation like in Daedalus.

- **Mapping and Scheduling**
The mapping and the scheduling problem are solved together with the design space exploration of the system-level architecture. The problems are put together to form one multi-objective optimization problem, whose solution gives answers to the sub-problems. To solve the optimization problem, an improved Strength Pareto Evolutionary Algorithm (SPEA2) [68] is used. The final mapping result is written in an XML file for the Espam tool to do code generation.

Since the KPN model of computation does not rely on the scheduling of processes, two simple scheduling algorithms are directly supported by the framework, First Come First Serve (FCFS) and Round-Robin. So far, there is no information about how applications with realtime constraints are handled in Daedalus.

- **Modeling and Simulation**

  The architecture modeling and simulation in Daedalus are done at high-level in the Sesame tool. The topology of the MPSoC platform can be configured in the tool by the user concerning which components should be taken from the IP library, how they ought to be connected, how many memory blocks should be instantiated, etc. The goal of the Sesame tool is to do automatic design space exploration, and the configuration of the best candidate platforms will be generated as result.

  In order to find the best candidates, high-level simulation is employed in Sesame to estimate the performance of each configuration for comparison. Unlike ISS based simulators, Sesame uses the application event trace to do the simulation. The application events are obtained through executing the KPN model natively in the host machine [69]. There are three types of application events traced, namely, the communication events read/write and the computational event execute. The first two correspond to the communication channel accesses in the KPN model, and the execute event records the execution of high-level functions such as the computation of a DCT transformation. To estimate the overall performance of the application, the event trace is rescheduled in Sesame according to the process-to-PE mapping and the hardware configuration of the platform, the functionality of the application is not simulated in this process. So far, no detail is available about how the timing for each function such a DCT transformation is estimated in the framework.

- **Code Generation**

  The Espam tool is the overall system code generator of Daedalus, whose software generation function is the most interesting part here. The software code generated by it is composed of three parts, process behavior in C, communication/synchronization primitive and memory map. The C behaviors of the processes are synthesized from the KPN specification in the XML file. The communication/synchronization primitives are realized through the
read/write FIFO channel function calls. According to the communication channel information in the XML file, the generator determines the address of the accessed FIFO, and a list of all FIFO addresses are put into a global memory map file.

Overall, the Daedalus framework automates the whole design flow of the MPSoC platform, though some limitations may apply due to the high-level of automation. The automatic KPN generation only supports SNLAPs, manual input is still needed for applications which are not in the category. Moreover, the default scheduling support of the framework is relatively simple, no realtime scheduling service is reported so far. The current Daedalus software development flow seems to be enough for the processor components in its IP component library and the MPSoC designed by the framework itself. The applicability of the approach to other MPSoC platforms e.g. the OMAP platform, is still to be seen.

2.3.9 MPA

The MPSoC Parallelization Assist (MPA) tool ([70], [71]) is developed at IMEC, and its goal is to help designers mapping applications onto embedded multicore platforms and doing design space exploration. Similar to the HOPES framework, the MPA tool targets existing MPSoC platforms, and currently it supports the ARM11 MPCore [35] processor. Besides, x86 multicore processors such as the AMD Phenom [11] processor are also supported as native verification platforms.

An overview of the MPA framework is depicted in Figure 2.17. The starting point of the exploration flow of the MPA is the sequential C source code of the target application. It is profiled by source level instrumentation and execution on the target platform or an Instruction-Set Simulator (ISS). Based on the profiling result, the programmer needs to create an initial parallel specification (ParSpec) in order to let the MPA tool generate the first parallel version of the application and trigger the exploration loop. The generated parallel C code is then simulated in a high-level simulator (HLsim), which additionally takes the profiling data and a target architecture specification as input. The result is produced as an execution trace, according to which the programmer then adapts either the architecture specification and/or the ParSpec for the next iteration of the exploration loop. Once the programmer is satisfied with the parallelization result, the result parallel C code can be directly reused on the target platform provided that the required runtime environment (RTlib) is already implemented on it.

- **Programming Model**
  The programming model of each single application in the MPA is still sequential, because it requires the input to be written in the C programming language. Since C is until today the most used programming language in the embedded area, and the MPA tool targets embedded MPSoCs, supporting C is therefore quite natural.

- **Parallelization**
  Along with the sequential C source code, the programmer needs to provide an additional file which explicitly specifies the application parallelism, i.e. the so-called ParSpec file. The ParSpec file controls the parallelization of the sequential input by specifying the computational partitioning of the application. The partitioning is defined by one or more parallel sections, outside a parallel section the code is sequentially executed, whereas inside a parallel section all code must be assigned to at least one thread. Both functional and data-level parallelism are supposed to be supported by the ParSpec.
Figure 2.17: Overview of the MPA Framework

Figure 2.18: MPA Application Example

Figure 2.18 gives an example, in which the sequential C source code is partially shown in a), and the corresponding ParSpec file is displayed in b). As it can be seen in the example, the parallel sections are recognized through labels in the source code. For instance, a parallel section `parsection1` is defined, whose source code is enclosed by the labeled block `parsect` (line 3 to 11), and the thread `T1` includes the code from the label `pr` to the end of the for loop (line 5 to 10). Data-level parallelism is explored through distributing different iterations of loops to multiple threads, e.g. the `T1` thread executes the `for` loop (line 7 to 10) with the index variable `i` ranging from 0 to 50 and the `T2` thread covers the range from 50 to 100. At the end of each parallel section, synchronization is implicitly done between the parallel threads and the master one. Figure 2.18 c) shows the relation of the threads in the example.

The combination of sequential C and separate parallel specification provides a flexible way of specifying different parallelization schemes. The same labeled statements and blocks can be grouped in different ways, and they can be stored in different ParSpec files without changing the C source code. However, the problem of how to come up a reasonable partitioning still needs to be solved by the programmer himself.

- **Mapping and Scheduling**

At the moment, the mapping and the scheduling of the parallel threads are managed by the target platform OS if available. The MPA runtime library, RTlib, is supposed to be implemented on top of the native thread functionality such as Pthreads, through which the
thread-to-processor mapping and the scheduling is handled dynamically by the OS. If the target platform does not have OS support, it is mentioned that a fixed mapping will be used, but no detail is so far available about how it is done.

- **Code Generation**

  The MPA tool generates parallel C code according to the input sequential C code and the ParSpec file. The thread model used by the parallel version is not the one directly provided by the target platform. Instead, it uses the API provided by the RTlib. Before the code generation, scalar dataflow analysis is performed on the functions containing parallel sections and those direct or indirect invoked callee functions. The result of the analysis is the so called Factored Use-Def (FUD) chains [72]. Base on the FUD chains, the code generator identifies data which need to be communicated across thread boundaries. Communication primitives are then inserted to the parallel C code, which use FIFO style communication channels for transferring the data. For some shared variables such as arrays, which are not directly handled by the tool, the designer needs to explicitly declare them as *shared* in the ParSpec file so that the code generator will generates the corresponding synchronization code.
• **Modeling and Simulation**

The parallelized application can be executed either on the target platform or on the HLsim which is a high-level simulator. The latter is mainly used by the designer for the purpose of quickly evaluating different parallelization schemes. In order to estimate the execution time, the HLsim uses the profiling information collected from the target platform or a cycle-accurate ISS. The delay introduced by communicating data between threads is computed according to the size of the data and the platform parameter which is specified in the architecture specification. After the simulation, the HLsim generates an execution trace, which is then analyzed by the designer to decide whether changes are needed for the application partitioning or not.

Overall, the focus of the MPA tool is on the parallelization of sequential C applications. Since the parallel sections in the C code and the parallel specification need to be manually created by the developer, the whole design flow still require a lot of user interactions. Besides, the tool completely relies on the target OS to do mapping and scheduling, and the framework has little control on both processes. Due to this reason, currently, no real time support is provided by the MPA tool.

### 2.3.10 TCT

TCT is the name of an MPSoC platform [73] which is developed at Tokyo Institute of Technology. It has its own programming model called Tightly-Coupled-Thread (TCT), which is an extension of C language. Figure 2.19 gives an overview of the complete framework. The design flow starts with the sequential C code with the thread extension. A special compiler is provided by the framework to compile the C code. After the compilation, the developer has two possibilities to execute the application, either in an ISS or on the TCT MPSoC hardware.

The TCT MPSoC uses one RISC processor and an array of six TCT co-processors as its processing engines [74]. The former features an ARM compatible instruction-set and works mainly as a controller, and the latter is designed for computation intensive applications. The co-processor array

![Figure 2.19: TCT Framework Overview](image-url)
has a fully distributed memory architecture, i.e. no memory is shared between co-processors. Each co-processor has its own memory and the communication is realized through dedicated channels, which are physically implemented through a crossbar switch.

- **Programming Model**

  The TCT programming model is designed with the goal of allowing programmers to write parallel applications in C code. From the coding point of view, its syntax is very simple. The developer just needs to insert thread constructs, which are similar to macros, into the C code. An example TCT C code is shown in Figure 2.20 a). By the developer, three threads are defined, which are $T_1$ (line 3 to 16), $T_2$ (line 7 to 10) and $T_3$ (line 11 to 14). Besides, there is a main thread which is always instantiated by the tool. In total, the example application has four threads executing in parallel, and Figure 2.20 b) illustrates the activation relation of them. Most C statements can be included in the TCT threads, with the exception that `goto` statements and dynamic memory allocations are not allowed.

  The execution model of TCT is a program-driven MIMD (Multiple Instruction Multiple Data) model where each thread is statically allocated to one processor ([75]). Since multiple flows of control are executed simultaneously, different parallelism schemes such as functional pipelining and data-level can be modeled. For programmers, the TCT programming model is very friendly in the sense that the inserted thread scope annotations do not change the sequential form of the C code much.

- **Parallelization**

  The TCT programming model provides an intuitive way for declaring code blocks to be executed in parallel. Nevertheless, the framework itself mainly relies on the developer to insert the parallel constructs explicitly. Hence, the exploration of the parallelism in the application is driven by the programmer.

- **Mapping and Scheduling**

  It is already mentioned earlier that each TCT thread is statically allocated to one processor. In this scenario, the mapping and the scheduling is very straightforward, because no special exploration process is needed.

```
1 int main() {
2    ... 
3    THREAD(T1) {
4       /* T1 body */
5       for (i=0; i<100; i++) {
6          ... 
7          THREAD(T2) {
8             /* T2 body */
9            ... 
10         }
11       THREAD(T3) {
12          /* T3 body */
13          ... 
14     }/* End for */
15     }/* End T1 */
16 }
```

Figure 2.20: TCT C Code Example
• **Code Generation**

Since the communication between threads is not explicitly specified in the C code, it is the job of the TCT compiler which analyzes the data transfer between threads by using the dependence flow graph [76]. Afterwards, the compiler inserts the inter-thread communication/synchronization instructions and generates the binary executable.

The communication and synchronization between threads are realized through special instructions, which are Control Token (CT), Data Transfer (DT) and Data Synchronization (DS). The CT instruction activates the execution of a thread; the DT instruction transfers data through a communication channel; and the DS instruction is used to check the availability of certain data and stall the thread if necessary until the data is available.

• **Modeling and Simulation**

An ISS is provided by the TCT framework for both software development and the design of the TCT MPSoC. Since it is also used for design space exploration, the number of processors that are instantiated in the simulator is configured according to the thread definition in the application. The simulator can do both instruction accurate simulation and trace driven simulation. In the latter case, the instruction behavior is not simulated, and therefore the simulation speed can be very high. However, in order to obtain an execution trace, the application has to be executed in the instruction-accurate model at least once.

The TCT framework provides a complete platform covering both software development support and MPSoC hardware. Compared to other frameworks, it supports a SW developer friendly programming model, which is enabled through the sophisticated control/data flow analysis built in the TCT compiler. Therefore, later in this thesis, the platform is used as target for developing new parallelization techniques.

2.4 **Summary**

MPSoC software development is a very challenging research topic. Due to the pressure of the market, the approaches followed by industrial companies are typically conservative in order to be more acceptable by developers who are similar with traditional software development flows. On the other side, academic groups are more active in proposing new programming models and languages and try to solve the problem from the top of the flow. However, almost no work has been done which is able to cover all aspects of the MPSoC software development problem including programming model, parallelization, mapping/scheduling, code generation and modeling/simulation. It can be seen that a lot of space is still there for new research works.
Chapter 3

Methodology Overview

3.1 MPSoC Application Programming Studio (MAPS)

MAPS is an MPSoC software development framework which aims to realize the methodology proposed by this thesis. The one and only goal of MAPS is to efficiently support MPSoC software development. Nevertheless, due to the complexity of the problem, it is unlikely that a single tool can provide a complete solution. Therefore, MAPS features a set of tools each of which

![Diagram of MAPS IDE and its components]

Figure 3.1: Overview
focuses on part(s) of the overall MPSoC programming process. Figure 3.1 shows an overview of the framework. With MAPS, the development of MPSoC software is carried out in a sequence of systematic steps. In the rest of this section, an overview will be given on each step in the tool flow.

3.1.1 Application Modeling

Generally speaking, an application model in the MAPS framework is a description of the application which should be deployed on the target MPSoC platform. As an input of the tool flow, it is supposed to contain all information that is necessary for driving MPSoC software development. Typically, this includes the application behavior, the application real-time requirements and the relation between multiple applications when they co-exist in one MPSoC. Since some of the information is only required when multiple applications are handled, MAPS separates the modeling of applications into two parts, namely, single-application and multi-application.

Single-Application Model

The MAPS single-application model focuses on individual applications when they are considered standalone. This includes their functionality and real-time behavior.

To describe the functionality of an application, a programming model is required. As it is already discussed in Chapter 1, two types of programming models are normally used by developers for this purpose: sequential and parallel. The C programming language is an example of the former, and it is until today still the most used programming language in the embedded domain [77]. The Message Passing Interface (MPI) [21] is an example of the latter, which has been used for decades in the high performance computing area. Although parallel programming models are well suited for applications which have intrinsic parallelism, none of them has been widely used in embedded devices.

The solution MAPS envisions, is an extension of the C language. The reasons for extending an existing language are twofold. On the one hand, since developers have written a huge amount of code in C during the past few decades, it is helpful to allow the reuse of these code in the MPSoC era. On the other hand, extensions are required to allow programmers to explicitly express parallelism when they know how the application should be parallelized. The model of computation supported by the extension should be flexible, so that most applications can be modeled without restriction. Currently, the Kahn Process Network (KPN) model [47] is targeted by the MAPS C extension. Moreover, the extension needs to allow programmers to directly specify real-time requirements with the MAPS extended semantic elements, so that no external data structure is required and the model inconsistency problem can be avoided.

Multi-Application Model

For devices which run only one application, the single-application model is enough for its software development. However, in reality, most devices such as mobile phones have more than one application running at the same time. In such an environment, scheduling and mapping tasks solely based on the knowledge of each individual application is not enough, because the processing elements are shared and the change for one application can easily influence the behavior of another.
Therefore, an overview of all applications in the target MPSoC is required for better coordinating their execution. In MAPS, this is provided by a multi-application model.

In principle, the MAPS multi-application model mainly describe the relationship between applications. For example, when multiple applications share one hardware resource such as a hardware accelerator, then only one of them can be executed at a time. This exclusive resource usage information is helpful for the scheduler to schedule these applications. Moreover, some applications are unlikely to run at the same time, like a video player normally does not run with an on-going teleconference. Although, technically, they can run in parallel, it is better for the device to spend more resources on the application with a higher priority. Such concurrency information should also be included in the multi-application model. In summary, the goal of the multi-application model is to store all information which is relevant to the global scheduling and mapping of applications.

### 3.1.2 Architecture Modeling

Heterogeneous processing elements and special communication architectures are often employed in MPSoCs to improve performance and/or energy consumption for a specific application or an application domain. The corresponding software development process must be aware of the MPSoC architecture so that the hardware resources can be efficiently utilized. For example, the knowledge of the instruction set architecture (ISA) of a processor can be used to estimate the computational cost of a piece of C code; an auto-parallelizing compiler could better partition the input program if it is aware of the difference between the processing elements in the target platform; task scheduling and mapping need to first know which and how many processors are there in the platform before any decision can be made; and a code generator needs to know for which MPSoC the code should be generated.

In MAPS, such knowledge is provided by an architecture model describing details of the processing elements and communication architectures of the target MPSoC platform. It serves as a centralized database as depicted in Figure 3.2, from which tools can query architecture information that is required for the MPSoC software development. Structural information such as the number and the type of the processing elements and their connections is provided for tools to have an overview of the topology of the target platform. Details like the instruction set of processing elements and the latency of communication channels should also be available to support the estimation of the computational/communication cost of MPSoC software.

Typically, the modeling of an MPSoC platform needs to be done only once, and the resulting model can be reused for different applications. Therefore, in the MAPS tool flow, the overall effort for creating architecture models needs to be kept as little as possible.

![Figure 3.2: Architecture Model](image-url)
3.1.3 Profiling

In software engineering, profiling is a commonly used technique for the investigation of software behavior. Normally, programmers use the profile information to get an overview of the application and identify the hotspots (most frequently executed parts of applications) which require more optimizations. Since profile information is collected during the execution of the program, profiling is also known as a form of dynamic analysis method.

As software development projects are nowadays carried out by a team of programmers, one developer can be asked to work on the code written by another. In MPSoC software development, where code reuse is emphasized, such situation happens even more often. Therefore, in MAPS, profiling is used as the first analysis process which can help programmers to get familiar with the application source code. Moreover, for the MAPS framework itself, the collected dynamic information is used by the control/data flow analysis process in the tool flow.

The profiling approach followed by MAPS uses the fine-grained source code instrumentation technique which is first introduced in micro-profiler [78], and an overview of the process is shown in Figure 3.3. From the source code of the input application, a special native program is created through instrumentation, whose execution will produce a trace file recording the execution history of the application. Afterwards, the trace file is post-processed together with the architecture model to produce an architecture aware profile for the target application.

3.1.4 Control/Data Flow Analysis

Control/data flow analysis is used in almost all compilers to get information about how a program executes and uses data. The control/data flow analysis in MAPS has a similar goal as its counterpart in traditional compilers. Moreover, it not only analyzes the dependence between different parts of source code, but also combines the resulting dependence information with dynamic profiling information in order to obtain a comprehensive view of the target application.

In traditional compilers, control/data flow analysis is static, i.e. the analyses are performed purely on the source code of the application without execution information. Consequently, the results of the analyses can only tell about the existence of control or data dependency. From static analysis, it is difficult sometimes impossible to get information about the amount of occurrences of the dependency which is useful for parallelizing the application. Therefore, dynamic profiling information is used by MAPS in its control/data flow analysis to complement the dependence information with runtime statistics. The result of the analysis process is generated in form of flow graphs whose nodes are fine-grained C level atomic operations such as addition, pointer

![Figure 3.3: Profiling Process](image-url)
dereference, etc. Such information is needed by the MAPS partitioning tool to search for the parallelism in the target application. From the user’s perspective, the analysis process is internal and completely transparent. It is triggered automatically before the MAPS partitioning tool is started.

### 3.1.5 Partitioning

The goal of the partitioning process is to find out the parallelism in the target application. The partitioning result is crucial for MPSoC, because only by separating an application into parallel tasks, is it possible to exploit the computational power provided by the multiple processing elements in MPSoCs. Typically, the code being partitioned is a complete sequential application. Besides, when the target application is modeled using a coarse-grained parallel programming model in which each parallel process is sequential and contains a lot of computations, code partitioning can be applied on the parallel process to further explore its internal parallelism. Traditionally, such partitioning work is done by developers manually. Unfortunately, this makes the development of MPSoC software tedious and error-prone. Therefore, a tool is provided by MAPS to help MPSoC programmers partition the application.

To partition an application into parallel tasks, there are three main issues need to be solved: granularity, dependency and parallelism. First is the granularity of the tasks, i.e. the amount of computations which should be performed by a task. Fine-grained tasks means that each task is small and can be finished within a short period of time. On the contrary, a coarse-grained task requires more computational time. The choice of granularity depends on the underlying MPSoC architecture and the behavior of the target application. Besides, the amount of communication between tasks also plays an important role. The fewer, the better. Ideally, when no data needs to be transferred between two tasks, they can be executed in parallel.

The second issue in the partitioning process is the dependence between tasks. In principle, tasks depend on each other mainly due to two possible causes: control and data. The control dependence exists when a task controls the activation of one or several other tasks, and the data dependence typically occurs when a task reads data which is produced by another task. The control and data dependence unveil the correlation between tasks. For instance, when a task produces a large amount of data for another task, they are closely related and might be better merged into one task to avoid the transfer of the data.

The available parallelism in the application is the last issue in partitioning. The biggest question to be answered in this place is which part(s) of the target application can be parallelized. Normally, control intensive code are unfriendly to parallelization, and loops are good candidates for constructing parallel tasks. It is challenging for a software tool to find out the hidden parallelism in applications.

The MAPS partitioning tool addresses the above mentioned three problems through a semi-automatic approach. An overview of the partitioning process is given in Figure 3.4. Given the dependence information provided by the previous control/data flow analysis process, the partitioning tool first automatically searches parallel task candidates on different granularity levels in the application. Afterwards, the programmer can choose to accept the generated result or review/modify it according to his knowledge about the application and the target MPSoC architecture.
3.1.6 Scheduling & Mapping

In MPSoC software development, the goal of mapping is to allocate processing elements to parallel tasks, i.e. on which processing element a task should run. It is also known as spatial mapping. Once the mapping information is available, the execution order of tasks on each processing element is then determined by scheduling. Scheduling is often referred to as temporal mapping.

Task scheduling and mapping are two highly correlated problems. The problem of which one shall be solved first, is very similar to the phase coupling problem in traditional compilers, where the order of instruction scheduling and register allocation needs to be determined. Since task scheduling must consider the real time requirements which can be stringent, it is not always possible to find a feasible schedule if the processing element has too many tasks mapped. In such case, if task mapping is performed first, then the result of task mapping needs to be reconsidered according to the scheduling result and some tasks might be remapped and rescheduled. The whole process continues until the application requirements are met.

Due to the high correlation between the task mapping and scheduling problems, MAPS targets solving them as a whole with one process. Instead of mapping and scheduling parallel tasks of one application individually, multiple MPSoC applications are considered altogether with respect to their requirements on the computational resources and real-time constraints. The MAPS multi-application model plays an important role in this place, since it provides information of all the applications in an MPSoC platform. The operating system running on the platform has to be considered, because the resulting task schedule needs to be realized on top of it. Moreover, for platforms without an operating system, a customized scheduler can be required to implement the task temporal mapping on processing elements.

3.1.7 Code Generation

With the application partitioned into parallel tasks and the mapping and scheduling of the tasks determined, the code generation process finally generates target code for the underlying MPSoC.
3.1. MPSOC APPLICATION PROGRAMMING STUDIO (MAPS)

Platform. The target code refers to source code which can be compiled for the processing elements on the target MPSoC. In most cases, C code is generated, because it is supported by most embedded processors. However, in special cases when no C compiler support is available, it can be necessary to generate assembly code for the target processor.

During the generation process, communication and synchronization primitives need to be added by the MAPS code generator to the generated target code. Typically, such primitives use the support provided by the operating system running on the target MPSoC. However, it is also possible that a processing element does not provide any such support. Then, the code generator is responsible for generating code to implement the required communication and synchronization functionality. Besides, some MPSoC platforms have their own support of parallel programming models, like MPI [21] and OpenMP [22]. For them, it is necessary for the MAPS code generator to be fully aware of the underlying programming model, so that the produced code can make the best use of the target platform.

3.1.8 High-Level Simulation

Traditionally, programmers need to use a real hardware prototype for testing the functionality of MPSoC software. Recently, Virtual Platforms (VPs) which are simulators of the target MPSoC platform, are more and more used by developers. VPs employ instruction set simulation technologies for running the instructions of target processing elements, and target binaries of MPSoC software can be executed on VPs without modification. Typically, less time is needed to develop a VP than a hardware prototype, because everything is simulated with software in VPs. However, since hardware details like instruction sets are required to be simulated in VPs and a lot of effort is required to create simulation models for such details, the development of VPs is still a time-consuming work. Often, MPSoC programmers still have to wait for VPs to be finished, which prevents the early test of MPSoC software and potentially reduces the efficiency of the development team. Therefore, high-level simulation is provided by MAPS as the means for early functional test of MPSoC software.

Since details of the target MPSoC platform might not be available at an early design stage, the architecture of the target MPSoC needs to be generalized and abstracted. For this purpose, a generic MPSoC simulator is provided by MAPS. Figure 3.5 roughly shows the whole simulation process. Programmers can easily configure it to have different numbers of processing elements and performance characteristics in order to let the simulator mimic the architecture of the target

![Figure 3.5: High-Level Simulation](image-url)
MPSoC. To do functional test, MPSoC softwares are natively executed in the simulator. To debug the source code, the programmer can simply use a native debugger. All these can be done at a very early design stage when no hardware prototype and VP are available with the high-level simulation support provided by the MAPS framework.

3.1.9 Target MPSoC Platform

The MAPS high-level simulation facility enables early testing of MPSoC software, but it is not enough to finally determine if the developed code fulfills criteria like real-time constraints, etc. This requires an accurate execution environment which cannot be easily available through highly abstracted simulation. Only a real hardware prototype or a VP with an accuracy at clock cycle level can provide the required accuracy and environment. Therefore, a target MPSoC platform is necessary in the MAPS tool flow as the final execution platform for software testing purposes. Except for the functionality of the application, the application timing must also be verified against the design criteria, when real time constraints exist.

Depending on availability, either hardware or VP can be used as test vehicle. Ideally, both kinds of platforms should behave the same and provide identical information to the programmer. Nevertheless, in reality, a VP might simplify some hardware details such as cache for the sake of simulation speed, the timing information provided by VPs might not be fully correct. Therefore, the most accurate timing information can only be derived from hardware.

3.1.10 Integrated Development Environment

As it can be seen from the previous sections, the complete MAPS tool flow consists of a number of processes. Some of them are automatic and do not require user interaction, e.g. control/data flow analysis; some work semi-automatically under the control of user like the MAPS partitioning tool. In order to create a fluent software design flow for MPSoCs, the tools need to be integrated together and put under the control of one design environment. For this purpose, the MAPS IDE (Integrated Development Environment) is designed.

The MAPS IDE is a software development environment based on the Eclipse [14] platform which is very popular in industrial companies for constructing customized design environments. The C/C++ development tools [79] from Eclipse are reused in the MAPS IDE to provide basic support for developing software in C. On top of it, the IDE is enhanced with a set of special MAPS plugins for controlling the execution of the tools involved in the MAPS design flow. The results produced from the tools are also visualized in the MAPS IDE for the programmer. Through integrating the tools in the MAPS IDE, a unified working environment is provided to MPSoC software developers.

3.2 Contribution of This Thesis

Within the scope of this thesis, the overall concept of the MAPS programming methodology is developed, and a set of tools are implemented to realize the proposed methodology. In this first implementation of the MAPS methodology, different processes in the MAPS development flow are realized as follows:
3.2. CONTRIBUTION OF THIS THESIS

- **Application Modeling**
  Since the MAPS single/multi-application models are still under development, sequential C source code is used here as the input application model to drive the development of the framework.

- **Architecture Modeling**
  A high-level modeling method is developed for creating abstract MPSoC architecture models.

- **Profiling**
  A trace based profiler is used to generate not only a source level performance profile for the programmer but also dynamic information for the control/data flow analysis.

- **Control/Data Flow Analysis**
  An analysis process is implemented, which combines static and dynamic analysis techniques for discovering the control and data dependences inside the application.

- **Partitioning**
  With the dependence information given by the previous process, a semi-automatic code partitioning tool is provided to partition a sequential application into parallel tasks.

- **Code Generation**
  For generating target code, a code generator is implemented, which is capable of automatically producing code for the TCT MPSoC platform [75]. For other MPSoC platforms used in this thesis, the code generation is done manually.

- **High-Level Simulation**
  The high-level simulation is done in this thesis with a generic MPSoC simulator which simulates MPSoC architectures at a very high abstraction level.

- **MAPS IDE**
  Last but not least, a prototype of the MAPS IDE is also constructed within this thesis to provide a convenient environment for using the above mentioned tools.

To prove the feasibility of the MAPS methodology, several MPSoC platforms are used in this thesis as target:

- **TCT**
  The TCT MPSoC platform [73] is developed in Tokyo Institute of Technology. The TCT hardware is composed of one RISC processor as main control and 6 co-processors as processing engine. This thesis uses the TCT MPSoC simulator to run the corresponding target code, which can be configured to have a processor number larger than 6. Besides, no operating system support is provided by TCT.

- **Multi-ARM**
  The multi-ARM platform is an in-house MPSoC platform [80] which consists of a configurable number of ARM926 [81] RISC processors. Besides, for the dynamic scheduling of software tasks, the platform features a special hardware block. Since no hardware prototype is available for the platform, VPs are built to run applications.

- **LT-RISC/VLIW**
  The LT-RISC/VLIW platform is an experimental MPSoC platform whose components are
the LT-RISC and LT-VLIW processors from the IP library of the CoWare processor designer [52]. To experiment different architecture configurations, VPs are built with different numbers and combinations of the processors. Like the TCT platform, the LT-RISC/VLIW platform also has no operating system available.

Several multimedia and signal processing applications are developed using the MAPS methodology. Since the target platforms provide limited support for scheduling, the mapping and scheduling are done manually for the test applications. The results show that the MAPS framework can support the programming for these platform efficiently. In the following chapters, the components which are implemented in this thesis will be discussed in detail.
In Chapter 3, it is mentioned that the development of MPSoC software is carried out by the MAPS framework in several steps. Within this design flow, the target architecture information is used by different processes. For example, the partitioning tool needs to know how many clock cycles it takes a processor to execute a piece of code, so that the tool can decide if the code should be parallelized or not; the mapping process needs to find out the most suitable PE for the parallelized tasks by taking into account their execution cost on different PEs. Due to the extensive use of the architecture knowledge, it is necessary for the framework to put all required information into a centralized location, which is the MAPS architecture model to be discussed in this chapter.

MAPS contains a high-level architecture model of the target MPSoC platform. It provides information about target processing elements for estimating the computational cost of software. Both homogeneous and heterogeneous architectures can be supported. Physically, the model is specified in XML format so that future improvements can be incorporated.

The remaining of this chapter is organized as follows. First, in Section 4.1, a brief analysis of the relation between performance estimation and architecture model is given to reason the modeling method used in MAPS. Afterwards, some related works in this field are discussed in Section 4.2. Then, the processing element model is introduced in Section 4.3. Following the discussion of the modeling methods, Section 4.4 presents how the model information is stored in the framework. Finally, the chapter is concluded with a short summary in Section 4.5.

### 4.1 Performance Estimation and Architecture Model

In the MAPS framework, the main usage of the architecture model is to do performance estimation. Generally, there are three types of approaches which are used to obtain performance information of an application or a piece of source code written in high-level programming language such as C:

- **Hardware execution** is the most accurate way to measure the software run time. However, since the creation of a hardware prototype requires a large amount of effort, it is not always possible for programmers to have one with them.
- **Simulation** is often used as an alternative approach instead of hardware. The speed and the accuracy of simulation heavily depend on the underlying simulator. On the one hand, using an RTL level simulator like ModelSim [82] can achieve very high accuracy, but the simulation speed of several tens of KIPS (Kilo Instruction Per Second) would give programmers a hard time for doing software development. On the other hand, high speed instruction-set simulation techniques introduced by companies like CoWare [52] can achieve several hundreds MIPS (Million Instructions Per Second) of simulation speed with the sacrifice of accuracy. Compared to hardware, simulation is able to give results relatively early. However, this approach often suffers from long execution time for large applications.

- **Abstract** models describe the underlying architecture on a very high abstraction level. It is proposed to provide a much more efficient performance estimation method with low modeling and execution effort [83]. Since no execution is done in this approach, the speed of estimation can be very high.

Table 4.1 summarizes the differences between the three above-mentioned approaches. In the MAPS design flow, processes like partitioning often have a number of design alternatives to be evaluated, e.g. a piece of code can be partitioned in different ways. The main criterion of making the decision is often the effect on performance. Since the number of the design alternatives can be very large, it is not possible to execute the software each time the performance information is needed. Therefore, MAPS employs a high-level approach for early performance estimation, and an abstract architecture model is developed and used.

<table>
<thead>
<tr>
<th>Method</th>
<th>Abstract</th>
<th>Simulation</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Level</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Low</td>
<td>Model dependent</td>
<td>100%</td>
</tr>
<tr>
<td>Speed</td>
<td>Very High</td>
<td>Model dependent</td>
<td>Realtime</td>
</tr>
<tr>
<td>Setup Effort</td>
<td>Low</td>
<td>Moderate</td>
<td>Very High</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of Performance Estimation Methods

In the next section, some existing abstract modeling methods for the performance estimation will be discussed.

### 4.2 Related Work

In [84], Giusto and Martin propose a method to estimate execution time of software running on a microprocessor. The target processor is modeled as virtual processor instruction set. The number of executed virtual instructions is counted by the *Cadence Virtual Component Co-Design* (VCC) [85] compiler and estimator. A set of benchmark applications are executed in an *Instruction Set Simulator* (ISS) of the target processor to get the real execution time. These collected data is then used in the multiple linear regression to estimate the execution cycles of virtual instructions at the CPU/Compiler level. After that the overall performance can be calculated according to the number of virtual instructions in an application. Due to the linear regression, the proposed method is accurate only when the benchmark applications and the application to be estimated are
similar. To define this similarity, a discriminator is developed, which tells whether the predictor can be used for other programs or not. The reported error range is between -12% to +5% when the estimated application is similar to those used for training.

A dynamic data analysis driven technology, which characterizes the performance of embedded applications, is presented by Bontempi and Kruijtzer in [86]. In comparison to the above mentioned method, a nonlinear function is used as estimator to predict the performance of applications. To calibrate the estimator, some representative training applications are required. Their performance measurements are first obtained through using a cycle accurate simulator, and then the parameters of the non-linear estimator are calculated by machine learning techniques. With this method, the reported mean error is about 8.8%.

The above introduced methods are improved in [83]. A more precise method using neural network to estimate the virtual instruction execution cycle is applied, which is mainly for advanced architectures. The results have errors up to 17% compared to the cycle accurate simulation.

Varbanescu et al. introduce a performance predictor for MPSoC with the name of PAM-SoC in [87], which is based on PALMELA [88]. The models of system (number of processors, memory behavior) and applications (instructions) are combined in this tool-chain in order to get timing information. The reported average error is about 19% for Wasabi platform [89], which consists of several DSPs and one or more general purpose processors.

The performance estimation method employed in MAPS, especially how the computational cost is calculated, is very similar to the approach which is introduced in [90]. A cost table which describes the execution cost of primitive C level operations such as addition, subtraction, etc, is used to characterize a target processor. In the following sections, details about how processing elements are modeled in MAPS will be discussed.

### 4.3 Processing Element Model

A complete MAPS architecture model is mainly comprised of two parts, Processing Element (PE) model and PE interconnection information. The PE model contains information about the computational resources available in the target MPSoC. As mentioned earlier, the provided information are at a high abstraction level; instead of directly specifying latencies of assembly instructions, C operation costs are described. To explain how this can be used to estimate the computational cost of a piece of C code, it is necessary to first have a look at the compilation process of a high level programming language like C.

#### 4.3.1 C Compilation Process

Typically, to program a processor, software developers can use both high-level programming languages, e.g. C/C++, and low-level language, i.e. the assembly. Today, programmers mostly use high-level languages due to their ease of use, productivity and portability across platforms. But even if a program is written in C, it must be first translated into low-level assembly language, which is, in principle, a symbolic representation of the numeric machine codes and constants. This translation process (compilation) is done by a compiler.

Generally speaking, a compiler is a program that translates a program written in one language
(the source language) into a semantically equivalent representation in another language (the target language). Over the years new programming languages have emerged, the target architectures continue to change, and the input programs become ever more ambitious in their scale and complexity. Thus, despite the long history of compiler design, and its standing as a relatively mature software technology, it is still an active research field. However, the basic tasks that any compiler must perform remain essentially the same. Conceptually, the compilation process can be subdivided into several phases as shown in Figure 4.1:

- **The frontend** creates an Intermediate Representation (IR) of the source program. The process can be further separated into several phases: the **lexical analysis** first breaks the input source text into a stream of meaningful tokens; then, the **syntax analysis** checks if the sequence of the tokens is correct according to a given grammar; finally, the **semantic analysis** verifies if the logic of the input program conforms to the rules which are defined in the programming language.

- **The middle-end** typically applies a sequence of high level, typically machine independent optimizations to transform the IR into a form that is better suited for code generation. This includes tasks such as common subexpression elimination, constant folding, constant propagation etc. A very common set of high level optimizations is described in [91].

- **The backend** constructs the target program in assembly language. Like the frontend, the backend is also composed of several phases: the **code selection** chooses suitable assembly instructions for the IR operations; then, the **register allocation** decides the mapping between variables and physical registers; afterwards, the instruction scheduling determines the order of the instructions; finally, the **code emission** outputs the assembly instruction stream to a text file which can be further processed by an assembler.

From the compilation process, it can be seen that IR and assembly both contain information of the input program, and hence can be used for estimating its computational cost. Theoretically, direct estimation from C statements is also possible. A comparison between them is shown in Figure 4.2. A C statement can contains a sequence of operations, e.g. multiplication and addition. In the IR, these operations are recognized and stored in a suitable data structure. The example IR is in the format of three-address-code, where each statement contains a maximum of one operation and all C operators and the majority of memory accesses are visible. IR operations are translated by the backend into assembly instructions. The translation is not always straight forward, e.g.
DSPs often support Multiply-Accumulate (MAC) instructions which can do a multiplication and an addition operation together.

Since a C statement can contain multiple calculations, as the example shows, using it directly for performance estimation could oversee the operations it carries. In comparison, assembly code is highly machine dependent; and a complex compiler backend is needed to get information about the generated instructions. For the purpose of MAPS, it is desired that the model used for performance estimation can be easily modified for new processor architectures. Therefore, instead of C statement and assembly, IR level information is used in the MAPS architecture model, which is kept in PE operation cost tables.

In comparison to using assembly, using IR for performance estimation is less accurate, because the effect of Instruction Level Parallelism (ILP), which is processor dependent, cannot be taken into account. Nevertheless, for RISC like architectures, a reasonable accuracy can be achieved [90]. Presently, this approach is chosen to drive the design of the remainder of the MAPS framework, and it is possible to improve the approach in future.

### 4.3.2 Operation Cost Table

In the MAPS architecture model, an operation cost table specifies the cost which is needed by a processor to finish C level IR operations. Mostly, the number of clock cycles is used as the measure of cost, but the user can also use other performance related numbers like time to fill the table instead.

#### 4.1. Definition (Operation). An IR level operation $o_{o,d}$ is a calculation, where

- $o \in O$, $O$ is the set of IR operators; and
- $d \in D$, $D$ is the set of data types.

Given the definition of the IR operation, a operation cost table can be defined as:

#### 4.2. Definition (Operation Cost Table). An operation cost table $C$ is a two dimensional array $C = [c_{o,d}]$, where $c_{o,d}$ denotes the estimated cost of one execution of the operation $o_{o,d}$.

The set of IR operators is defined by the compiler frontend which produces the IR, and the data types are defined by the C language specification. For example, the value of $c_{\text{add}, \text{char}}$ in a cost table is the estimated cost of the target processor doing an addition calculation on two character variables. In the IR, calculation with high-level composite data types like `struct`, are typically transformed by the frontend into a sequence of operations on primitive data types like pointer and
integer. Figure 4.2 shows an example of the transformation of a `struct` member access, which is done by the frontend of the LANCE compiler [92]. The simple assignment statement whose target is a member in a `struct` is lowered into pointer and integer arithmetics. Since the IR operations are closer to assembly instructions, using them for computational cost estimation is more accurate than using C statements directly.

The cost values are typically derived from the instruction cycle times of the target processor architecture, which can normally be found in its technical reference manual. However, since the mapping between IR operators and assembly instructions is not always straightforward, some entries need to be determined through compilation. Table 4.3 shows an operation cost example of the ARM9EJ-S processor [93]. According to its manual [81], its addition operation takes 1 cycle to finish in most cases. Here, the bit width of `int` and `long` is set to 32, which is the same as the register file width of the processor. Therefore, the estimated cost of their addition is set to 1. Nonetheless, since the data of the `long long` type has a bit width of 64, larger than an ARM9EJ-S processor can directly compute with one instruction, and the compiler translates one 64bit addition into two 32bit addition instructions, the value of 2 is used for the corresponding entry in the table.

<table>
<thead>
<tr>
<th>Operator</th>
<th><code>int</code></th>
<th><code>long</code></th>
<th><code>long long</code></th>
<th><code>...</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Example Operation Cost of ARM9EJ-S

The cost table approach can well support RISC processors [78]. Since the target platforms in this thesis mostly feature RISC like instruction-sets, the approach is sufficient. However, for processors with instruction-level parallelism such as VLIW processors, the accuracy achieved by the approach is not enough and further improvement is necessary, which is not covered by this thesis.

### 4.3.3 Computational Cost Estimation

Based on the operation cost table, the overall computational cost $C_{overall}$ of a C program can be calculated by the following equation:

$$C_{overall} = \sum_{\forall o \in O} \sum_{\forall d \in D} c_{od} \times n_{od}$$

(4.1)

$n_{od}$ in the equation is the total number of the operation $o_{od}$ performed in the program.
An example C program is shown in Figure 4.3, which contains only one function, and no control flow statement like if/then/else is used. Its IR code and the related operation cost and number values are given along with the C code. Since the program is very simple, its computational cost can be easily be calculated, which is \( C_{overall} = 7 \).

```c
int main()
{
  int a=1, b=2, c;
  c=a+b;
  return c;
}
```

**C Code**

```c
int main()
{
  int a_3, b_4, c_5, t1;
  a_3 = 1;        //load const.
  b_4 = 2;        //load const.
  t1 = a_3 + b_4; //add
  c_5 = t1;       //assign
  return c_5;     //return
}
```

**IR Code**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load const.</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Assign</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Return</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.3: A Simple Example of Computational Cost Estimation

Note that, to illustrate the calculation of the cost, the above example is kept as minimum, and does not contain any loop or other structure which can change the execution flow, and hence counting operations directly in the IR is possible. In reality, applications contain complex control flows whose execution can depend on input data, i.e. applications behave differently according to the input data they get. This makes the static cost estimation from IR nearly infeasible. Therefore, MAPS employs an advanced approach which combines static analysis and dynamic information gathered from execution. Its details will be discussed in Chapter 5.

### 4.4 Architecture Description File

In MAPS, an architecture description file is used to centrally store all the information about the PE model and the architecture of the target MPSoC. For the rest of the framework, it serves as a database from which the architecture information can be retrieved for the estimation of computational cost. In order to make the description easily extensible, XML (Extensible Markup Language [94]) is used as the format for the architecture description, which is currently the de facto standard for information exchange, and future improvement can be incorporated into it by extending the specification.

```xml
<mapsarch:Architecture xmlns:mapsarch="MAPSARCH">
  <ProcessingElement>...</ProcessingElement>
  ...<!— PE Description —>...
  <Processor ...>...</Processor>
  ...<!— Processor Instances —>...
  <Channel ...>...</Channel>
  ...<!— Inter-Processor Connections —>...
</mapsarch:Architecture>
```

Figure 4.4: Overall Structure of the MAPS Architecture Description File

Figure 4.4 shows the overall structure of an architecture description file. The information contained is mainly composed of three parts:
• **PE description** mainly contains the cost tables of the processing elements used in the target MPSoC platform;

• **Architectural description** describes the overall architecture of the MPSoC, which tells how many processors are available in the MPSoC, and how they are connected.

The ProcessingElement nodes are the PE models, and the architectural description is contained in the Processor and Channel nodes. In the rest of this section, these two parts of the description will be introduced respectively.

### 4.4.1 Processing Element Description

In Section 4.3, it is already discussed that PEs are modeled in MAPS using operation cost tables. Therefore, the PE description mainly contains the table for the corresponding processor. Besides, since MPSoCs can employ different PEs, a ProcessingElement node is needed for each type of PE, and an architecture description file must have at least one PE description.

An example of the PE description is shown in Figure 4.5. In each ProcessingElement node, there are two sub-nodes defined:

- **CoreType** specifies the type of the described PE. Processing elements with the same type can refer to the same ProcessingElement node for the model information. Besides, there must be one CoreType in a ProcessingElement.

- **CostTable** saves information of different operation costs. Similar to the CoreType node, its existence in ProcessingElement is also mandatory. The basic element in CostTable is Operation. The execution cost of the Operation with different VariableType is contained in the Cost node.
  
  - **Operation** contains the cost for one specific operation with different types of variable. The type of operation is stored in its attribute Name.
  
  - A VariableType node corresponds to one type of data which can occur in the compiler IR. Since C is the programming language supported by MAPS, normal C data types like int and char can appear in this node. The Name attribute of the node contains the type name.
  
  - **Cost** is the leaf node of the whole PE description. It gives the execution cost of an operation with a specific variable type. The value of Cost is typically integral. For generality, floating point values can also be used here.

The Figure 4.5 example shows that this PE description is for the ARM9EJ-S processor core, and the cost for a addition operation with the data type of int is 1, i.e. \( C_{\text{Add}, \text{int}} = 1 \).

### 4.4.2 Architectural Description

The PE description provides part of the information about an MPSoC, which is required for the computational cost estimation for each individual PE. Information about the overall structure of the target platform is stored somewhere else, which are the Processor and Channel nodes in the architecture description XML file.
4.4. ARCHITECTURE DESCRIPTION FILE

Figure 4.5: PE Description Example

- A Processor node represents one instance of a PE in the target MPSoC. An architecture description must contain at least one PE instance. The type of the PE is specified in the Type sub-node, which contains a text string referring one of the PE descriptions specified in the current XML file. When multiple PEs of the same type are integrated in one MPSoC, they are distinguished from each other through the ID attribute of the node, whose content must be a unique identifier.

- A Channel node represents one connection between two PEs. The type of the connection, e.g. DMA, is specified in the Protocol sub-node. The Source and the Target attributes specify the two end points of the channel, which are PEs. Since it is possible that multiple channels using different communication mechanisms like DMA or bus, can co-exist between two PEs, multiple channels are allowed between them. An ID attribute is used to identify different channel instances.

Figure 4.6 shows an example architectural description of the SHAPES RISC DSP Tile (RDT). From the XML data, it can be seen that the RDT is composed of two PEs, one ARM9EJ-S processor and one mAgic processor. The PEs are connected through a DOL (see Section 2.3.5) communication channel.

It needs to be mentioned that communication channels are not covered by the current architecture model. Their models are still under development and will be added to the MAPS architecture model in future.

![Architectural Description Example](image-url)
4.5 Summary

Modeling MPSoC architectures is itself a research topic, which can be very complex. The MAPS architecture model abstracts the target platform to a very high-level. The proposed approach uses compiler IR level operation costs to describe how efficient one PE is in doing computation. The model information is stored in an extensible form so that there is enough space for future improvements. The complete specification of the format is given in Appendix A.

The modeling work is a one time effort, one platform only needs to be described once and the result can be reused for different applications. For the other parts of the MAPS framework, the created architecture model is taken as an input together with the application model.
Profiling is a common software analysis technique which is widely used today. In contrast to static code analysis, profiling uses information gathered from program execution. Therefore, it is a form of dynamic program analysis. In the MAPS framework, it is employed as the first analysis process. This chapter introduces the details of the MAPS profiling tool.

The goal of the profiling process in the MAPS framework is twofold. First, it provides source level profile information to MPSoC software developers to help them with the analysis of the application behavior, which is useful when the source code is obtained from a third party resource. Second, the dynamic information is internally used by other processes in the MAPS framework, like control/data flow analysis.

To start profiling, two inputs are required, source code of the target application and an architecture model of the target MPSoC platform, which is created according to the specification given in Chapter 4. The whole process is carried out in two steps, trace generation and post-processing. In the trace generation step, the application is compiled by a special compiler toolchain and executed in the host machine. The execution of the specially compiled application will produce a trace file which records the important dynamic information which is interesting to MAPS. Afterwards, the post-processing tool analyzes the generated trace file together with the provided architecture model and produces the final profile data.

For the user of MAPS, the result of profiling is provided in two forms. A source level performance profile is given by the tool, which shows the estimated computational cost per each source line. Besides, a dynamic call graph is created, which unveils the calling relationship between functions along with their estimated computational costs. Both are visualized in the MAPS IDE for the user. For MAPS itself, the profile information is given at a much more fine grained level, i.e. compiler Intermediate Representation (IR) level, to be used internally for dynamic analysis.

The rest of this chapter is organized as follows: Section 5.1 first discusses some research works as well as industrial products which are related to profiling; then, an overview of the whole profiling process is shown in Section 5.2, which is followed by Section 5.3 presenting details about the code instrumentation procedure and the profiling runtime system, which are employed in this work; afterwards, Section 5.4 introduces the post-processing procedure in together with its results; finally, Section 5.5 briefly summarizes the whole chapter.
5.1 Related Work

Profiling is nowadays a widely used technique for analyzing softwares. A lot of works have already been done in this area. According to the profiling technique, profilers can be categorized into three categories, sampling based, simulation based and instrumentation based.

Sampling based profilers like [95] and [96], probe the program counter of the host processor at regular intervals by utilizing operating system interrupts. Since only an additional interrupt routine is injected to the operating system, the application binary is not modified, the approach is non-intrusive and has little performance loss. Nevertheless, due to the fact that sampling profilers do not monitor the events occurred between the intervals, their results are typically not very accurate and cannot be easily annotated back to the compiler IR of the application. Besides, memory access profiling is not possible with this approach.

Simulation based profilers are often at the same time instruction set simulators, e.g. [97]. Application binaries are executed in a completely monitored environment, and hence instruction level profile information can be obtained, which is very accurate and fine grained. However, because the target instructions are completely simulated with software, the performance is normally three orders of magnitude slower than native host execution; and the result is specific to the simulated instruction set architecture.

Instrumentation inserts additional instructions or code into the target program to collect the required information. By carefully inserting code to different places in the program, different runtime information can be collected. Therefore, instrumentation based profilers are flexible in terms of the result they generate. As a matter of fact, today, most profilers are based on code instrumentation techniques. Since code instrumentation can be done at different levels, they can be further classified into source, fine-grained and binary profilers.

Source code level instrumentation is the most widespread type of profiling technique, mostly for C/C++ code. Standard tools like GNU gprof/gcov [98] can be used to generate statistics about CPU time spent in different functions as well as execution frequencies of code lines. Programmers typically uses such tools to optimize the application code by rewriting the hot spots for more efficient execution on the target machine. The profiling information generated by source-level profiling is mostly coarse-grained. Depending on the coding style, a single line of C code can consist of a number of different operations that map to many assembly instructions. Besides, implicit address computations and memory accesses, pointer scaling, and type casts are not directly visible in high-level languages like C/C++. All are these are not captured by source-level instrumentation.

In order to improve the profiling result, fine-grained instrumentation is proposed. In [99], the SIT toolkit is presented which performs C-operation level profiling by exploiting the C++ operator overloading capability. However, this approach cannot recognize different C operators with similar instruction-level semantics. For example, pointer indirection (“*ptr”), as well as array (“[]”) structure access (“− >”) all map to “LOAD”/“STORE” instructions, but they are different operators in C++. In [78], a tool called micro-profiler is proposed, which instruments the target application at compiler IR level. At this level, the high-level operations like array element accesses are all lowered to primitive operations like memory load/store, the problems brought by the operator overloading are thus overcome. In fine-grained profiling, application source lines are broken down into small operations and instrumented. Therefore, more code is inserted than source-level profiling does, and hence the instrumented application typically runs slower. Moreover, both source and fine-grained profiling tools are specific to the programming languages which are
supported by the code instrumentation process.

Binary instrumentation tools like [100], [101] and [102], are not programming language specific, because they instrument application binaries dynamically at runtime. They are machine specific and difficult to retarget for new processor architectures. In the embedded area, where the diversity of Instruction-Set Architecture is large, and the variation of programming language is small, binary instrumentation is not widely accepted.

For the MAPS framework, which requires an accurate and retargetable solution, the fine-grained IR level profiling approach is used. The MAPS profiling process is similar to the work which is presented in [78], but differs in several aspects. First, instead of generating the profile information during the execution, profiling is done in two steps, trace generation and post-processing. Second, the architecture model is managed separately from the instrumented application, therefore, the post-processing can flexibly generate the profiling result for different processing elements without instrumenting and executing the application again.

5.2 Profiling Process Overview

Profilers execute the analyzed application in order to collect dynamic information, however, the statistic of the result is not always available directly after the execution. Two approaches are normally taken by developers when they come to this point, either calculating the statistics during or generating a trace file and post-process it. Both have their advantages and disadvantages. The first approach does the calculation at runtime, which influences the application execution speed. If the profile calculation is computationally intensive, e.g. cache profiling, the speed of the application can become unacceptable. Hence, it is suited only for profiling application characteristics which do not require much computational power like function call profile. The second approach postpones the computation of the final result by separating the whole procedure in two steps, trace generation and post-processing. During the execution of the application, the dynamic events which are interesting to the profiler, are recorded in a trace file. Afterwards, the file is post-processed which eventually produces the final profiling result. Since the computation of the profile is postponed, this approach is less intrusive than the former one. Some profilers even use both approaches at the same time. For example, the micro-profiler does function profiling at runtime, but generates memory traces for calculating the cache profile in a post-process afterwards.

In the MAPS framework, the second approach is employed. The profiling process is carried out in two steps, whose overview is shown in Figure 5.1. The input of the process is the C source code of the application. It is first instrumented and compiled by a special toolchain into native executable. The execution of the instrumented application will generate a trace file recording the events which are important for profiling. The trace file is then post-processed together with the MAPS architecture model to produce the final profiling result.

It can be seen that the profiling result is computed in the post-process; hence, the trace generation can be done independently from the target architecture. The architecture information is first used when the computational cost is calculated. This separation of the execution and the profile calculation provides the benefit that profiling applications for different PEs can be done without executing them again. In such cases, the micro-profiler, on the other hand, needs to compile and execute the application multiple times. Since the MAPS framework needs to deal with heterogeneous MPSoCs, the avoidance of multiple application execution makes the approach more favorable.
5.3 Trace Generation

The trace of an application typically contains records of events, which occurred during the execution. The decision of which events should be recorded depends on the interests of the profiler user. Since MAPS uses the runtime information for the estimation of computational cost and the analysis of dynamic data dependence, two kinds of information are recorded, fine-grained execution history and memory access. The former is realized by keeping the record of the sequence of the executed Basic-Blocks (BBs), and the latter is stored along with the BB trace.

To enable the generation of such traces, a set of tools are needed. In the overview diagram (Figure 5.1), it is shown that the LANCE compiler [92] frontend first takes the C source code as input. An Intermediate Representation (IR) file, which is in the format of 3-Address Code (3-AC), is generated by the frontend. Then, the instrumentation tool reads in the IR, inserts additional code (mainly function calls) to appropriate places and eventually generates a modified version of the application IR. Afterwards, the instrumented IR is compiled by the host compiler into a native executable program. The implementation of the instrumented function calls is provided by the profiling runtime library, which needs to be linked together with the application in the compilation process. In the following sub-sections, the whole procedure will be explained with more details.

5.3.1 LANCE Frontend

It is mentioned in the earlier section that MAPS profiles applications at compiler IR level, which is more fine-grained and accurate than source level profiling. To do this, a compiler frontend is needed to first transform the input C source code into IR. For this purpose, MAPS uses the frontend of the LANCE compiler framework [92]. The format of the IR used in LANCE is so called (IR-C) code, which is 3-Address Code written in the high-level C language.

---

1Basic-Block is a concept commonly used in compilers for analyzing source code. A basic block is a linear sequence of code with unique entry and exit points.[91]
In the IR-C code, all high-level C constructs, such as for/while loops, complex arithmetic expressions, and implicit address arithmetic for array and structure access are all transformed by the LANCE frontend into primitive statements. There are in total five different types of statements defined in the LANCE IR-C, which are assignment, branch, conditional branch, label and return statements. Functions in the IR-C code directly correspond to their original definition in the input C code. Each function has a local symbol table, which keeps information about local variables, and a global symbol table is defined in each IR-C file for storing information of global variables.

Figure 5.2 gives an example of the IR-C code, which is generated from the LANCE frontend. From the example, it can be seen that the for loop is lowered into several assignment statements with conditional branches; each assignment statement has at most one computational operation like compare or add.

In comparison to the high-level C, the IR-C code is primitive in terms of its realization of control/flow structure and simple statements. Since the IR-C code is still compliant to the C language, it can be easily compiled by a host compiler into native executable. Besides, the LANCE IR-C writer allows user callback functions during the IR-C generation. These features make it convenient for implementing code instrumentation and native application profiling.

### 5.3.2 Instrumentation

Based on the IR-C code delivered by the LANCE frontend, the instrumentation procedure adds C code to appropriate places for the profiling purpose. Functions are instrumented to record their entries and exits; BBs are instrumented so that their execution sequence can be kept; and, memory accesses are instrumented to generate memory traces for the dynamic data flow analysis in the framework. The inserted code are mainly function calls, which write data to the generated trace.
file and maintain some dynamic information like the current status of stack. The implementation of the functions is provided in the profiling runtime library, which needs to be linked with the instrumented application.

The code instrumentor is implemented by using the API of the LANCE framework. The IR-C code can be read/written with the API functions. Besides, the framework also provides iterators for functions, BBs and IR statements, which makes the traversal of the whole IR data structure very easy. The pseudo code of the code instrumentor is shown in Section B.1. The entrance of each function is instrumented with a call to the `EnterFunction` function. Then, a series of initializations are done for both global and local variables.

After the instrumentation per each function is done, the code instrumentor iterates through all the IR statements, and inserts code before those statements having special semantics. Except for the entries of BBs, which can be looked up from the BB list of the function, the instrumentor needs to check the IR operator inside the statement in order to understand its behavior. Additionally, special attention is paid to dynamic memory allocation functions like `malloc` or `free` so that heap objects can also be recognized at runtime. In the following section, the profiling runtime library will be introduced in detail.

### 5.3.3 Profiling Runtime Library

Since the goal of executing the instrumented application is to record dynamic information in a trace file, no complex statistics need to be collected at runtime, and hence the implementation of the profiling runtime library can be kept relatively simple and straightforward. Most functions in the profiling runtime library just write a record in the trace file for their corresponding events. For instance, upon the entry of a BB, the `EnterBasicBlock` function will write the ID of the BB in the trace file.

Nevertheless, the generation of the memory access trace is not as trivial as that of the BBs, because the recorded information is much more than an address. When an application executes, the addresses of variables can be quite dynamic. This is especially true for local variables, because when a function is called, a block of memory will be allocated in the stack for its local variables, register spills\(^2\) etc. After the function is finished, the memory block will then be released and used for other functions, which implies that accessing the same memory address does not mean accessing the same variable. In case if the profiled application uses `malloc` and `free` intensively, one memory block can be used multiple times during the execution for different purposes, it is difficult to deduce the accessed variable only based on an address. Therefore, memory addresses are first disambiguated at runtime before they are written into the trace file.

In order to be able to check the memory object one address points to, a dynamic symbol table is needed, which keeps the addresses of all variables at runtime. This is achieved through the variable initialization functions inserted at the beginning of each function. Addresses of local variables are written to the table each time a function is called; when the function exits, the addresses of its local variables are removed. Entries for global variables in the table are permanent, which are created at the beginning of the `main` function. Besides, when the application uses heap memory through `malloc` or `free`, the corresponding addresses will be inserted/removed to/from the table accordingly.

---

\(^2\)Memory locations for saving registers temporarily.
5.3. TRACE GENERATION

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnterFunction</td>
<td>Initialize function stack information and write the enter of function to trace</td>
</tr>
<tr>
<td>InitGlobalVariable</td>
<td>Add an global variable to the dynamic symbol table</td>
</tr>
<tr>
<td>InitLocalVariable</td>
<td>Add an local variable to the dynamic symbol table</td>
</tr>
<tr>
<td>EnterBasicBlock</td>
<td>Write the ID of the basic block to trace</td>
</tr>
<tr>
<td>TraceMemAccess</td>
<td>Look up the accessed address, and write the access information to trace</td>
</tr>
<tr>
<td>PrepareFunctionCall</td>
<td>Store the calling statement ID to a variable</td>
</tr>
<tr>
<td>TraceMalloc</td>
<td>Add an entry for the allocated memory object to the dynamic symbol table</td>
</tr>
<tr>
<td>TraceMalloc</td>
<td>Add an entry for the allocated memory object to the dynamic symbol table</td>
</tr>
<tr>
<td>TraceFree</td>
<td>Remove the entry of the released memory block from the dynamic symbol table</td>
</tr>
<tr>
<td>ExitFunction</td>
<td>Clean up the stack information of the exited function, remove the entries of the local variables from the dynamic symbol table, and write the exit of the function to trace.</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of Profiling Runtime Functions

The functions in the profiling runtime library perform the management of the dynamic symbol table and the generation of the trace file. A brief summary of the functions is given in Table 5.1. With these functions, the execution of the profiled application is kept track of with fine-grained details in the trace file, as will explained in the next section.

5.3.4 Trace File

The generated trace file contains three levels of information, function, basic-block and memory access, which give a detailed view on the execution of the profiled application. Figure 5.3 shows an example which is taken from the trace file generated from a JPEG encoder application. In the text format trace file, there are five types of trace lines:

```
s:0:enter:main:myJPEG5.c 240 # Enter function main
s:2584:enter:initializeWorkspace:myJPEG5.c 82 # Enter function initializeWorkspace
m:489|z|g|imageQuality|4|0|0
... # Read global variable imageQuality
m:375|w|l|jpeg_make_c_derived_tbl|1|huffsize|2|1 # Write local variable huffsize
... # Exit function initializeWorkspace
exit:initializeWorkspace:myJPEG5.c # Exit function initializeWorkspace
... # Exit function main
```

Figure 5.3: Example Trace File

- **Enter Function**: a function entry line tells the entrance of a function with its name and the ID of the calling IR statement and the source file where the function definition is. The format of the line is:
• **Exit Function**: a function exit line tells that the return of a function with its name and source location. The format of the line is:

```
exit:<Function Name>:<Source File Name>
```

• **Enter BB**: the BB entry line simply contains the ID of the entered BB.

• **Global Variable Access**: a global variable trace line gives information about the statement ID where the access occurs, the name of the variable, the access type (Read or Write), the accessed data type, and the offset of the access. The format of the line is:

```
m:<Statement ID>|<Access Type>|g|<Variable Name>|<Data Type ID>|<Offset>|\<Last Define Statement ID
```

Note that, since the LANCE IR-C is used, the accessed data type is recorded using the type ID defined in the LANCE compiler framework. Besides, the ID of the last IR statement which defines the variable is also written into the trace file to ease the dynamic dependence analysis.

• **Local Variable Access**: a local variable trace line is very similar to that of the global variable, but with several additional items, which are the name of the function defining the variable and its instance ID. Since a function can be invoked multiple times and recursively, i.e. there can be multiple instances of the same local variable alive at the same time, the instance ID is then used to identify the accessed local variable exactly. Each time a function is entered, the number of total calls to this function is used as the instance ID and kept in its stack information. The format of the generated trace line is:

```
m:<Statement ID>|<Access Type>|l|<Function Name>|<Instance ID>|<Variable Name> \<Data Type ID>|<Offset>|<Last Define Statement ID
```

The information kept in the trace file is detailed. From the function enter/exit information, the dynamic call graph of the profiled application can be easily created. Besides, based on the complete execution sequence of the basic-block IDs, it is possible compute statistics on the executed IR statements as done in the post-processing step.

### 5.4 Post-Processing

The trace generated by the execution of the instrumented application provides the dynamic information of it, but in a raw format. The execution history is kept as a sequence of BB IDs with marks of function entries and exits, and memory accesses are recorded as individual trace lines. Since data in such a form is indigestible for the user of MAPS, the trace file must be post-processed in order to extract useful information.

An overview on the post-processing procedure is given in Figure 5.4. First, the trace file is analyzed together with the IR of the application source code and the target architecture model. The result of the analysis is an IR profile, which contains fine-grained profiling information about each IR
statement. Based on the information at a fine-grained IR-level, the dynamic call graph and the source profile of the application are then generated and presented to the user.

The IR profile is used for not only the generation of source level profile, but also other internal analysis which needs dynamic information, like the dynamic data flow analysis to be introduced in Chapter 6. The rest of this section mainly focuses on the creation of the IR profile and its use for the call graph and the source profile generation.

5.4.1 IR Profile Generation

The IR profile generated by MAPS contains application dynamic information which is fine-grained at the level of compiler IR statement. Since the input trace file only provides information at basic-block and function level, it is necessary to have the original IR of the application as an additional input. Besides, a description of the target MPSoC architecture in the form of the MAPS architecture model, is also used here as input so that the characteristics of the target platform can be taken into account.

With the given inputs, the generator first computes the intrinsic cost of each IR statement of the application. The term intrinsic means that the cost is caused by the statement itself. In case a call statement invokes another function, its intrinsic cost does not include the cost of the invoked function which can only be determined with dynamic information. Before the definition of the statement intrinsic cost is given, the IR of an application is first defined here as:

5.1. Definition (Application Intermediate Representation). The intermediate representation of an application is a triplet, \( IR = (S, B, F) \), where:

- \( S \) is a set of the IR statements;
- \( B \) is a set of the basic-blocks; and
- \( F \) is a set of the functions.

Besides, the set of processing elements in the target MPSoC platform is defined here as \( P \), which can be known from the architecture model.
5.2. Definition (Statement Intrinsic Cost). \( \text{StmtIntrinsicCost}(s, p) \) is the estimated cost of executing \( s \) once on \( P \), where:

- \( s \in S \), if \( s \) is a call statement, the cost of the callee function is excluded; and
- \( p \in P \).

The application IR used here is the LANCE IR, which has already been used in Section 5.3 for code instrumentation. Since the LANCE IR statement is in the form of three-address code, and one statement contains at most one operator, the calculation of \( \text{StmtIntrinsicCost}(s, p) \) can be easily done by first checking the type of the operator and the operand data type and then looking up the cost value in the corresponding operation cost table (see Section 4.3) in the input architecture model. The process is very straightforward. Based on the statement intrinsic cost table, the BB intrinsic cost is computed as next, which is defined as:

5.3. Definition (BB Intrinsic Cost). \( \text{BBIntrinsicCost}(b, p) \) is the estimated cost of executing \( b \) once on \( p \), where:

- \( b \in B \), if any functions are invoked in \( b \), their cost is excluded; and
- \( p \in P \).

Since a BB is simply composed of a sequence of IR statements with at most one statement changing the control flow at the end, \( \text{BBIntrinsicCost}(b, p) \) can be calculated by adding the statement intrinsic costs, as presented in the following equation:

\[
\text{BBIntrinsicCost}(b, p) = \sum_{s \in b} \text{StmtIntrinsicCost}(s, p) \tag{5.1}
\]

Note that, BBs are numbered with unique IDs at the beginning of the post-processing by the same numbering scheme as the one used in the code instrumentation for trace generation. Therefore, by counting the BB IDs in the trace file and checking the function call boundaries, which are marked with the enter and exit trace lines, the execution count of a BB in each function call can be obtained. Since a function can be invoked from different places in the application and behave differently, it is necessary to use the IR statement which invokes the function, i.e. call site, to distinguish different instances of function calls.

5.4. Definition (BB Execution Count). \( \text{BBExecutionCount}(b, f, c) \) is the number of times that \( b \) is executed when its parent function, \( f \), is invoked by \( c \), where:

- \( c \in S \), \( c \) is the caller IR statement;
- \( b \in B \), and \( f \in F \).

The above definition of the BB execution count does not differentiate multiple calls to the same function done by the same IR statement when the application runs, although the invoked function can have different behaviors. In the context of MAPS, this is not required, because the IR profile aims to provide an overall statistics on the computational cost contribution of each IR statement in the application. Multiple function calls from one location are, therefore, counted together. As next, the generator computes the self cost of functions.
5.5. Definition (Function Self Cost). FunctionSelfCost(f, c, p) is the estimated cost of executing f on p, when f is invoked by c, where:

- \( f \in F \), if functions are invoked from f, their cost is excluded;
- \( c \in S \), c is the caller IR statement; and
- \( p \in P \).

Given \( BB\text{ExecutionCount}(b, f, c) \) and \( BB\text{IntrinsicCost}(b, p) \), FunctionSelfCost(f, c, p) can be calculated by the following equation:

\[
\text{FunctionSelfCost}(f, c, p) = \sum_{\forall b \in f} BB\text{ExecutionCount}(b, f, c) \ast BB\text{IntrinsicCost}(b, p) \tag{5.2}
\]

In contrast to the function self cost, the function call cost, FunctionCallCost(f, c, p), takes the cost of the callee functions into account.

5.6. Definition (Function Call Cost). FunctionCallCost(f, c, p) is the estimated cost of executing f and its callees on p, when f is invoked by c, where:

- \( c \in S \), c is the caller IR statement;
- \( f \in F \), and \( p \in P \).

From the definition, it can be seen that FunctionCallCost(f, c, p) = FunctionSelfCost(f, c, p), when f does not invoke any subroutine, which is a special case. Generally, the calculation of FunctionCallCost(f, c, p) must check each IR call statement in the function and calculate the callee function cost. Section B.2 shows the pseudo code which computes the function call cost.

For a statement c which calls f, FunctionCallCost(f, c, p) is the cost which is introduced by the statement to its parent function. Together with the intrinsic cost of c and its execution count, which can be deduced from the execution count of its parent BB, it is possible to define the overall execution cost of IR statements.

5.7. Definition (Statement Execution Cost). StmtExecutionCost(s, p) is the estimated cost of executing s and the function it may call on p, where \( s \in S \) and \( p \in P \).

StmtExecutionCost(s, p) characterizes the overall contribution of s in the whole application, no matter where its parent function is invoked, the cost is counted together. To get the overall execution count of an IR statement s, it is necessary to sum its parent BB’s execution count, BBExecutionCount(b, f, c) for all call sites where their parent function is invoked. The pseudo code of the general procedure which calculate the execution cost of IR statements is presented in Section B.3.

Similar to the statement execution cost, which does not differentiate function call sites, the overall estimated execution cost of each function is also calculated.
5.8. Definition (Function Total Cost). \( FunctionTotalCost(f, p) \) is the estimated cost of executing \( f \) and the functions it may call on \( p \), where \( s \in S \) and \( p \in P \).

Since the cost for each call site of \( f \) is already given as \( FunctionCallCost(f, c, p) \), the computation of \( FunctionTotalCost(f, p) \) can be easily done by adding \( FunctionCallCost(f, c, p) \) together, as shown in Equation 5.3, where \( c \rightarrow f \) means that \( c \) invokes \( f \).

\[
FunctionTotalCost(f, p) = \sum_{c \rightarrow f} FunctionCallCost(f, c, p)
\] (5.3)

The IR profile generated in this step provides detailed dynamic information about the application at different levels including IR, BB and function. For a programmer, the large amount of raw numbers can be too detailed to be understood. Therefore, further post-processing is needed to generate information which is readable for the user of MAPS.

5.4.2 Source Profile Generation

Based on the statistic information in the IR profile, a source level profile is created in the post-processing procedure, which tells the programmer the estimated execution cost of each line of C code in the application. Here, the line execution cost is defined as:

5.9. Definition (Line Execution Cost). \( LineExecutionCost(l, p) \) is the estimated cost of executing the source line \( l \) and the functions invoked in \( l \) on \( p \), where:

- \( l \in L \), \( L \) is the set of all source lines in the application; and
- \( p \in P \).

With the previous generated IR profile providing the IR statement execution cost information, i.e. \( StmtExecutionCost(s, p) \), the overall execution cost of one line of C code can be computed by calculating the summation of all the costs of related IR statements, as shown in Equation 5.4, where \( s \in l \) means that the IR statement \( s \) is generated from source line \( l \). This connection between IR statement and source location is typically created by compiler frontends for debugging purposes. The source profile generator of MAPS utilizes the debug information available in the LANCE IR.

\[
LineExecutionCost(l, p) = \sum_{s \in l} StmtExecutionCost(s, p)
\] (5.4)

The final line execution cost information is stored in an XML file, which will be visualized by the MAPS IDE for the programmer. Figure 5.5 shows part of a source profile XML file, which is obtained by profiling an JPEG application. From the example, it can be seen that a relative percentage of the execution cost of each line of code is also calculated by the generator and stored along with the estimate cost. The complete specification of the profile XML file is given in Appendix C.

Figure 5.6 shows a screen shot of the MAPS IDE visualizing the Figure 5.5 example together with the application C source code. The source profile information is displayed beside the corresponding
line numbers, from which the programmer can directly see the estimated cost of the code and thereby find the hot spot quickly. In the example, it can be seen that line 1004 is the most computation intensive part of the code. Note that, the source profile generated in MAPS takes the callee cost also into account, which is differently handled by the micro-profiler [78]. The micro-profiler does not consider the cost of the function which is invoked by a line of C code. Therefore, line 1003 and 1004 would be considered to have the same execution cost by the micro-profiler.

The cost information from the source profile tells the programmer how much computation is required behind each line of code, and an overview on the distribution of the computation cost within a function is also provided.
5.4.3 Call Graph Generation

In addition to the source profile, the post-processing of the trace file produces also a dynamic call graph for the MAPS user, which describes the exact calling relationship between functions during the execution. Moreover, based on the cost information provided by the IR profile, the generator also annotates the estimated cost of functions to their corresponding nodes in the call graph. By using this information, the programmer can quickly identify the computation intensive functions for further optimization or parallelization.

5.10. Definition (Dynamic Call Graph). A dynamic call graph is a weighted directed graph \( G = (F, E, p) \) with

- \( F \): the node set, in which each element, \( f \in F \), represents a function and is labeled with three values, \( W_{\text{count}}(f) \), \( W_{\text{self}}(f) \) and \( W_{\text{exec}}(f) \);
- \( W_{\text{count}}(f) \): the number of times that function \( f \) is called;
- \( W_{\text{self}}(f) \): the overall self cost of function \( f \);
- \( W_{\text{total}}(f) \): the total execution cost of function \( f \);
- \( E \): the set of directed edges, and each edge, \( e \in E : e = (x, y) \), represents that function \( x \) calls \( y \);
- \( W_{e}(e) \), \( e = (x, y) \): the edge weight which represents the number of times that \( x \) has called \( y \); and
- \( p \in P, p \): the target processing element;

Note that, the call graph created by MAPS is context insensitive, i.e. if a callee function is invoked by several caller functions, there is only one node instantiated in the graph for the callee. The value of \( W_{\text{self}}(f) \) takes all call sites of the callee into account and unveils how much computation is totally performed locally inside the function. The function self cost, \( \text{FunctionSelfCost}(f, c, p) \) which is directly provided by the IR profile data, however, differentiate each call site. The value of \( W_{\text{self}}(f) \) can be easily computed from \( \text{FunctionSelfCost}(f, c, p) \). The pseudo code for generating the call graph from trace is given out in Section B.4.

After the generation is finished, the resulting call graph is eventually stored in form of an XML file for later being visualized by the MAPS IDE. Figure 5.7 shows a piece of the call graph XML file of the same JPEG application used in Figure 5.5. The \( \text{Function} \) element in the example tells that there is a function \( \text{BLK8x8} \) in the application, which is executed for 2850 times. Its estimated self cost is 85500 and its total cost is 68768716 which corresponds to 57% of the overall estimated cost. The big difference between the self and the total cost indicates that the function itself is small, but the function(s) it calls consumes a lot of time. In the example call graph, the \( \text{Call} \) element denotes the caller/callee relation between two functions, and it can be seen that the \( \text{DCTcore} \) function has been invoked 2850 times by the \( \text{BLK8x8} \) function during the execution.

Additionally, the LANCE IR provides information about the source code location of the functions, and this information is also stored in the XML file. The location information is used by the MAPS IDE to quickly navigate the source code editor to the function which is selected by the user in the call graph viewer. Figure 5.8 shows part of the visualized call graph of the above example,
from which the relation ship between the function \textit{BLK8x8} and other functions can be easily seen. It can be seen that the function \textit{BLK8x8} is only invoked by the function \textit{JPEGtop} and invokes several other functions except for the \textit{DCTcore}. For convenience, the cost information is also displayed at the same time in the graphical view for the user.

From the dynamic call graph provided by MAPS, the user can quickly obtain an overview of the functions unveiling their calling relationship and estimated cost. For MPSoC software development, especially parallelization, it is often that those most time consuming or computation intensive functions are the focus of development.

### 5.5 Summary

This chapter introduces the profiling method employed by the MAPS framework, which involves a trace generation and a post-processing procedure. Through native execution, the former generates
a trace file containing not only the application execution history but also high-level memory access information. The post-process then refines the raw information and provides a source-level profile and a call graph to the MAPS user to help analyzing the application. Internally, an IR profile is created for generating the source-level information and use in the other part of the framework, e.g. the control/data flow analysis process introduced in Chapter 6.
Chapter 6

Control/Data Flow Analysis

In traditional C compilers, control/data flow analysis is a standard component, which is responsible for the construction of a global “understanding” of the program, so that code optimization techniques like loop fission [103] can be implemented. Similarly, the control/data flow analysis in MAPS also analyzes the input C source code in terms of its internal control and data dependences, and the result is used by the partitioning tool for parallelizing the application. Nevertheless, while normal compilers do the analysis statically, MAPS employs both static and dynamic approaches. Due to the nature of the C programming language, especially the use of pointers, it is impossible to realize a 100% accurate static control/data flow analysis [23]. Therefore, runtime information is used by MAPS in parallel to static analysis to find the internal dependences including those caused by pointers. Thereby, the framework is able to get a better inside view of the target application for the later parallelization work.

The goal of the analysis introduced in this chapter is to create a so called Weighted Statement Control/Data Flow Graph (WSCDFG), which describes the target application from several aspects. First, WSCDFG is a flow graph whose edges represent the control and data dependences in the application. Second, the nodes and edges of the WSCDFG are annotated with values which are derived from the profiling information, so that the application runtime behavior is also captured. Moreover, the graph is constructed at IR statement level, which implies that the information contained in the WSCDFG is very fine-grained. The static analysis used in this work is based on the existing facility of the LANCE compiler framework. Since techniques for static analysis have already been well discussed in the literature, this chapter focuses on the dynamic part of the analysis process.

The rest chapter is organized as following: Section 6.1 first briefly discusses the traditional Control Flow Graph (CFG), which is commonly used in most compilers; Section 6.2 then explains the concept of Statement Control Flow Graph (SCFG), which is in principle fine-grained CFG; Statement Control/Data Flow Graph (SCDFG) is as a step further from the SCFG, and it is introduced in Section 6.3; the definition and construction of WSCDFG is eventually discussed in detail in Section 6.4. Finally, Section 6.5 briefly summarizes the whole chapter.
6.1 Control Flow Graph

The basic concept of the WSCDFG is very similar to that of the control/data flow graph, which is used in most compilers for describing the flow-of-control information of the program. For each function, the nodes of its control flow graph are basic blocks (BBs), and the edges represent the control flow between BBs.

6.1. Definition (Control Flow Graph). A control flow graph of a function, f, is a directed graph G = (V, E), where

- V is a set of nodes, in which each node v ∈ V, represents a basic block;
- E ⊆ V × V is a set of directed edges, in which each edge e = (v, v′) ∈ E, represents a potential control flow between v and v′, which indicates that v′ can be executed directly after v;
- v′ is a successor of v, and v is a predecessor of v′;
- ∃!entry ∈ V, entry does not have a predecessor;

Moreover, the set of all successors of a BB v is denoted as Succ(v), and the set of all predecessors of v is denoted as Pred(v), with:

- Succ(v) = { v′ ∈ V | (v, v′) ∈ E }
- Pred(v) = { v′ ∈ V | (v′, v) ∈ E } 

An example program is shown in Figure 6.1 a) with the LANCE three-address IR and the CFG of its main function displayed in Figure 6.1 b) and c) respectively. The structure of the function is very simple, there is only one if-then-else statement. The corresponding CFG has four BBs: entry, bb1, bb2 and exit, and four edges: (entry, bb1), (entry, bb2), (bb1, exit) and (bb2, exit). Note that, although there is no IR statement at the end of entry, which jumps to bb1, the edge (entry, bb1) still exists, because BBs are linearly arranged in the function and bb1 directly follows entry in the three-address IR. If the jump condition of the if statement fails, bb1 will be executed directly after entry by default, which realizes the control flow from entry to bb1.

From the example, it can be seen that CFGs describe the control flow of the program on basic block basis. Since BBs are defined according to the control flow, their size is highly variable. The IR statement sequence of a BB can be arbitrarily long, as long as there is no branch in between. Therefore, no matter how different the sizes of BBs are, in CFGs they will be considered to be equal. For instance, the bb1 in the example contains only one computational statement, but the bb2 contains calls to two functions which are externally implemented. Nonetheless, they are seen as equal blocks in the CFG. For the development of MPSoC software, especially parallelization, the view of considering all BBs to be equal hides the behavior of the IR statements and prevents the analysis inside the BBs and further intra-BB optimizations. In order to overcome this limitation, BBs are first broken down to IR statements in the MAPS framework for analysis. The graph which describes the IR statement level flow-of-control information, is called Statement Control Flow Graph, in short SCFG.
6.2 Statement Control Flow Graph

The definition of the SCFG is similar to that of the CFG. The main difference is that the nodes of SCFGs are IR statements instead of BBs as in a CFG.

6.2. Definition (Statement Control Flow Graph). A statement control flow graph of a function, \( f \), is a directed graph \( G = (S, E) \) with

- \( S \) is a set of nodes, in which each node \( s \in S \), represents an IR statement;
- \( E \subseteq S \times S \) is a set of directed edges, in which each edge \( e = (s, s') \in E \), represents a potential control flow from \( s \) to \( s' \), which means that \( s' \) can be executed directly after \( s \);
- \( s' \) is a successor of \( s \), and \( s \) is a predecessor of \( s' \);
- \( \exists! \) entry \( \in S \), entry does not have predecessor.

In principle, SCFGs can be seen as fine-grained CFGs, in which BBs are replaced by sequences of IR statements. Figure 6.2 shows the SCFG of the Figure 6.1 example. Note that, although labels do not contain any computational operation, they are kept as targets of branch statements. From the marked BB boundaries, it can be seen that the overall structure of the example SCFG is very similar to the CFG in Figure 6.1 c). Since there is no branch statement in the middle of BBs, the IR statements belonging to the same BB are simply connected by a sequence of control flow edges.

Due to the similarity between SCFG and CFG, SCFGs can be easily constructed from CFGs. The control flow analysis facility of the LANCE compiler is reused in the MAPS framework to generate CFGs first; then based on the LANCE CFGs, MAPS constructs the required SCFGs. Section D.1 shows the pseudo code which constructs a SCFG from a LANCE CFG.

In SCFGs, BBs are broken down into statements. However, there is only control-flow information in SCFGs. The data dependence information is not contained. Therefore, based on SCFGs, MAPS constructs so called Statement Control/Data Flow Graphs (SCDFGs) to carry both control and data dependence information at the same time.
6.3 Statement Control/Data Flow Graph

In SCDFGs, the edges are categorized into two classes, control edges and data edges. The control edges are completely inherited from SCFGs, which represent the control flow of the program. The data dependence information are represented by the data edges.

6.3. Definition (Statement Control/Data Flow Graph). A statement control/data flow graph of a function, \( f \), is an extended SCDFG \( G = (S, E_c, E_d) \) with

- \( S \) is the node set;
- \( E_c \) is the control edge set;
- \( E_d \subseteq S \times S \) is a set of data edges, and each edge, \( e_d = (s, s') \in E_d \), represents a data flow from \( s \) to \( s' \), i.e. the execution of \( s' \) will read data which is written by the execution of \( s \); and
- For a data edge \( e_d = (s, s') \), \( s \) is the producer, and \( s' \) is the consumer.

Note that, the above SCDFG definition only considers the read after write (RAW) dependency [91], which shows how data is produced and consumed by statements. There are two other kinds of data dependencies in computer programs, namely write after read (WAR) and write after write (WAW) dependency. In principle, the latter two indicate how data is overwritten in the program. For partitioning a application, the existence of a RAW dependence requires the data to be transferred from the producer to the consumer, but overwriting an old data does not. Therefore, in MAPS, where the control/data flow analysis is used for partitioning, only the RAW dependency is considered.

Besides, if a statement calls a function which implicitly modifies some data, such change of data is often referred to as side effect in compiler literatures like [104]. In the above definition of the data edge, the execution of \( s \) includes the behavior of the IR statement itself and the function it might
invoke, i.e. the side effect of \( s \) must be taken into account. Figure 6.3 b) presents the SCDFG of a simple program which contains such implicit data dependence. The dotted lines in the graph are data edges with the name of the variable which causes the dependency labeled beside. It can be seen in the example that, although \( s_2 \) and \( s_4 \) do not read or write any variable directly, the functions they call access a global variable \( g \). This dependence results in a data edge (\( s_2, s_4 \)) in the SCDFG. In comparison, the data edge (\( s_1, s_3 \)) is caused by a local variable \( a \) accessed directly in the IR statements.

Since a SCDFG can be seen as a SCFG plus some additional data edges, its construction can be done in a straightforward way, if a SCFG is given. Section D.2 gives out the pseudo code which constructs a SCDFG from a SCFG. The code relies on a function, GetProducer(\( s \)), to retrieve data producers of a statement \( s \). The implementation of this function requires dependence analysis, which is done here by using both static and dynamic methods.

### 6.3.1 Data Dependence Analysis

As mentioned at the beginning of this chapter, MAPS reuses the static data flow analysis facility of the LANCE compiler framework. Two functions are provided by the LANCE data flow analysis:

- \( \text{GetStatementUse}(s) \) returns the set of variables which are used by the statement \( s \); and
- \( \text{GetDefines}(v) \) returns the set of IR statements which define the variable \( v \).

Based on the two LANCE provided functions, the static part of the dependence analysis is implemented as a function \( \text{GetStaticProducer}(s) \), whose pseudo code is shown in Section D.3.1. Nevertheless, the data flow analysis implemented in the LANCE compiler is limited in that it only analyzes dependence among local scalar variables. For global variables and variables of aggregate data types like structure and array, no enough detail is provided [105]. Therefore, the dynamic analysis is used here in parallel to the LANCE analysis to find data dependence for all non-scalar and global variables.

The dynamic data dependence analysis uses the memory access trace which is generated in the profiling process. The raw format of the generated memory trace lines has already been introduced
in Chapter 5. In principle, from the trace lines, all details about a memory access are known, including the ID of the accessing IR statement, the type of the access, the target variable, etc. During the generation of the source profile and the dynamic call graph, these memory trace lines are simply ignored, because they are not needed there. However, in the control/data flow analysis here, they are used for the extraction of the dynamic data dependence information. A function $\text{GetDynamicProducer}(s)$ is constructed to retrieve the producer IR statements which are found in the memory traces. Section D.3.2 shows the pseudo code of the $\text{GetDynamicProducer}(s)$ function.

Although the SCDFG is capable of carrying both control and data flow information, its nodes and edges do not really reflect the amount of computation and data dependences which take place during the execution of the program. For instance, a statement inside a loop can be executed thousands of times, but a statement at the beginning of a function might run only once in the actual execution. In SCDFGs, they are considered to be the same, which can be misleading. Hence, the Weighted Statement Control/Data Flow Graphs (WSCDFGs) are introduced to incorporate profiling information so that a complete "understanding" of the program can be obtained.

### 6.4 Weighted Statement Control/Data Flow Graph

Generally speaking, WSCDFGs can be seen as annotated SCDFGs, whose nodes and edges are all annotated with values carrying specific meanings. Compared to the previously introduced graphs, WSCDFGs contain the most information which can be obtained from analysis, and hence are used by MAPS as the analysis output format to be further used in parallelization.

**6.4. Definition (Weighted Statement Control/Data Flow Graph).**

A weighted statement control/data flow graph of a function, $f$, is an annotated statement control/data flow graph, $G = (S, E_c, E_d, W_s, W_c, W_d)$, where

- $S$, $E_c$ and $E_d$ are the sets of nodes, control edges and data edges of the SCDFG;
- $W_s(s)$ is the weight of the IR statement $s$;
- $W_c(e_c)$ is the weight of the control edge $e_c = (s, s')$; and
- $W_d(e_d)$ is the weight of the data edge $e_d = (s, s')$.

As it can be seen from the WSCDFG definition, WSCDFGs are closely related to SCDFGs, and they differ only in the annotated node and edge weights. Therefore, the construction of WSCDFGs can be easily done on the base of SCDFGs. The detailed pseudo code which constructs a WSCDFG from a given SCDFG, is presented in Section D.4. Three annotation functions, $\text{GetStatementWeight}(s)$, $\text{GetControlEdgeWeight}(e_c)$ and $\text{GetDataEdgeWeight}(e_d)$, are used to get the weight values of the nodes and edges. Theoretically, the weight values can be calculated according to any arbitrary measure. Nonetheless, in the MAPS framework, the node weight values are related to the computational cost of the IR statements, and the edge weight values are computed according to the amount of control/data dependences occurred in the execution.

Figure 6.4 shows a WSCDFG which is constructed from the example of the Figure 6.3 b). The edge weight values are directly labeled beside the edges, and the node weight values are listed
on the right hand side of the diagram. In the example, the control edges are all labeled with 1, since the corresponding control flow occurred exactly once in the execution. Besides, the two data edges in the example are both labeled with the value of 4, because the variables which cause the dependence have the size of 4 bytes\(^1\). The value of \(W_s(entry)\) is zero, because a label does not perform any computation; and \(W_s(s_1)\) equals one, since the cost for assignment is one. In the following sections, the computation of the weight values will be discussed in detail.

### 6.4.1 Node Weight Annotation

The weights of the WSCDFG nodes, i.e. IR statements, are calculated from the overall computational cost of the statement. Remember, in Chapter 5, an IR profile is created which holds the profiling information of each IR statement, and a function, \(StmtExecutionCost(s)\), is provided, which returns the overall execution cost of a statement. The function is not only used for the computation of the source profile, the value it returns is also directly used here in the construction of WSCDFG as node weight, as shown in the pseudo code of Section D.4.1.

### 6.4.2 Control Edge Annotation

In the design of WSCDFG, the weight of the control edge is used to reflect the intensity of the corresponding control flow in the application. Its value is determined according to the number of times that a control flow is taken during the execution of the application. For instance, suppose there is a control edge \(e_c = (s, s')\) in a WSCDFG, each time the IR statement \(s'\) is executed directly after \(s\), the control flow from \(s\) to \(s'\) is taken. If \(s\) and \(s'\) are inside the same basic block, it is obvious that the total occurrence of their control flow equals the execution count of the BB. In case they are not in the same BB, their parent BBs must be executed in sequence in order to take the control flow. Therefore, by counting the number of occurrence that two BBs appear one after the other in the execution trace, the occurrence of the control flow from the last IR statement of the preceding BB to the first statement of the succeeding BB can be determined. Section D.4.2 is the pseudo code which implements the function \(GetControlEdgeWeight(e_c)\).

\(^1\)The size of an int variable is assumed to be 4 bytes.
6.4.3 Data Edge Annotation

The weight of the data edge reveals the amount of data dependences between two IR statements. In order to quantify the data dependence, the size of variable and the number of occurrence are used. The result is the product of the values both, and Section D.4.3 shows the pseudo code of the function, $\text{GetDataEdgeWeight}(e_d)$, which computes the weight of the given data edge. Since the size of variable can be easily looked up in the symbol table of the application IR, most of the code is used to find out the occurrence of the data dependence. As it is shown in the code, in case the producer and the consumer statements are in the same BB, the number of occurrence is the same as the execution of the BB. Nonetheless, when they are not in the same BB, it is necessary to first construct the set of BBs which define the dependent variable, and count the number occurrence from the trace file.

6.5 Summary

This chapter introduces the control and data flow analysis used in the MAPS framework. The final product of the analysis are so called Weighted Statement Control/Data Flow Graphs, in short WSCDFGs. WSCDFGs describe the control and data dependence at fine-grained IR statement level. Moreover, they incorporate profiling information through the weight of the graph nodes and edges, so that an overview is provided which covers not only the existence but also the amount of occurrence of the dependence relationship. Through WSCDFGs, the MAPS tools are able to know the computational cost of the application and the dependence within the application. It is mainly based on this knowledge, that the partitioning process finds the potential parallelism in the application. Chapter 7 discusses how the parallelism is found by the MAPS partitioning tool.
The MAPS partitioning tool discussed in this chapter splits a sequential application into parallel tasks. This process is also called parallelization and has been intensively researched in the high performance computing community for parallel compilers. These parallel compilers mostly do static analysis and focus on the parallelization of loops.

MAPS takes a different approach. First, the WSCDFGs produced by the previous analysis process are used to provide detailed information about the control and data dependence inside the application. Then, the partitioning process uses the information to find parallel tasks. A novel granularity concept, coupled block, is employed here in order to overcome the limitation of using basic blocks or functions as basic parallelization units. The process finds application parts which are closely connected through control and data dependences, and makes parallel tasks out of them. Finally, instead of doing everything automatically, the MAPS GUI presents the parallel tasks found by the partitioning tool to the programmer, who eventually decides if the machine produced result is good enough or not. Since the programmer possesses the application knowledge which cannot be easily analyzed by an algorithm or a software, human interaction is allowed in the MAPS partitioning process to keep flexibility.

The rest of this chapter is organized as follows: Section 7.1 first provides a survey about some related works in the area of parallelization; then, the granularity problem is discussed in Section 7.2; afterwards, in Section 7.3, the granularity concept, coupled block, is introduced; the algorithm which is used in the partitioning process to generate coupled blocks, is presented in Section 7.4; Section 7.6 gives details about how the user can refine the partitioning result; finally, Section 7.7 summarizes the whole chapter.

### 7.1 Related Work

Since the dawn of parallel computing, there have been plenty of works published about the automatic extracting parallelism from sequential applications. For instance, in [106], some general techniques are introduced to expose coarse-grained parallelism in C applications. These techniques in principle are complementary to the approach used by MAPS and can be integrated into the
Besides, several attempts to produce a partition of an application starting from a sequential implementation have been reported in literatures. [107] and [108] derive KPNs from loops based on the COMPAAN compiler technology. The derivation is possible for those Static Affine Nested Loop Programs (SANLPs) whose loop conditions are affine combinations of iterator variables. Because of this limitation, it would be difficult to apply the technique on applications with dynamic behavior and code with complex loop conditions.

In [109], an algorithm is presented, which parallelizes C applications through pipelining the execution of loops. The approach is similar to the one employed in the MAPS partitioning tool in that weighted flow graphs are also used to provide control and data dependence information. However, the analysis is performed solely on outermost loops and the partitioning method is simple and not capable of handling hierarchy such as nested loops and loops with function calls.

In [110], a HW/SW co-design environment is introduced, which targets reconfigurable systems for data-intensive applications. A three-step algorithm is employed in the environment to extract coarse-grained parallelism, which includes task clustering, partitioning and scheduling. Nevertheless, the entry point of the algorithm is a Directed Acyclic Graph (DAG) representation of the application, and manual effort is required for the conversion from implementations written in high-level programming language such as C to the DAG. Such conversion is normally nontrivial.

Similarly, [111] presents an approach in the network processor domain, which also uses dynamic analysis like MAPS does. Runtime instruction traces are used in the work to derive an Annotated DAG (ADAG) representation of the application, which is later taken by the clustering and mapping processes as input. In comparison to the MAPS dynamic analysis approach, applications are profiled on assembly level. It could be difficult to annotate the partitioning results back to the original source code as coarse-grained tasks, because the mapping between assembly code and source code is often broken due to compiler optimizations.

[112] proposes a designer-controlled code partitioning approach for system level MPSoC models. A set of six source level code transformations are developed in the framework to partition C code and data structures. Since the transformations are completely user-controlled, the programmer would needs to first know the target application before a reasonable partitioning can be created. In comparison, MAPS employs a semi-automatic partitioning approach which first automatically searches for parallel tasks and then let the programmer interact with the tool to determine the final parallel tasks.

In the high performance computing community, parallelization has been studied for decades. Although, scientific applications normally feature intrinsic parallelism and are easier to be parallelized than embedded multimedia applications, some works in that field are related to the MAPS approach in terms of the employed techniques.

For example, [113] proposes a Decoupled Software Pipelining (DSWP) approach to analyze the dependence graph of an application and generates a DAG out of a loop by merging together the strongly connected components which are parts highly dependent on each other. Thereafter, pipeline balancing is performed in a first-bin-packing fashion, and the solution that reduces the communication overhead is selected. This approach is targeted for loops and it is not clear if loops containing function calls are supported. Besides, the time budget for each pipeline stage in the loop must be provided by the programmer in advance, which could be not easy.

[114] presents an approach to exploit pipelined parallelism from C applications with user annota-
7.2 Task Granularity

To find parallel tasks in a sequential implementation, granularity is a major issue in the searching process and has a direct impact on the kind and degree of parallelism that can be achieved. Typically, fine-grained tasks perform small amounts of computations and can be finished within a short period of time. On the contrary, coarse-grained tasks do more computations and take longer to finish than fine-grained ones do. Theoretically, given the same amount of computations, more fine-grained tasks are required to finish the calculation than using coarse-grained tasks. Since parallel tasks need to communicate and synchronize with each other, extra time is needed to do inter-task communication and synchronization, which is not needed in a sequential application and is hence considered as overhead. Such parallelization overhead is most of the times inevitable. Even in the ideal case that no data needs to be communicated between parallel tasks, synchronization still takes place to coordinate the task execution. Therefore, using a large number of small parallel tasks normally means that the introduced parallelization overhead is also high. Sometimes, such overhead can be so high that no speedup can be achieved through using multiple processors, and it is not seldom that a parallel application runs slower than its sequential version.

Coarse-grained tasks have higher data locality and therefore synchronize with each other less often. Consequently, less parallelization overhead will be introduced when an application is coarse-grained parallelized. It is mainly due to this reason, that coarse-grained parallelism is favored by programmers for developing embedded applications. However, since there are the complex control and data dependences inside applications, it is not always possible to parallelize them by exploring coarse-grained parallelism. To find the appropriate granularity for parallelization, the application needs to be analyzed with respect to its internal control and data dependency.

Traditional compilers analyze applications written in high level programming languages on different granularity levels. The most commonly used constructs are:

- **Statement**
  In programming languages such as C, a statement typically includes one or a sequence of related calculations. It is the smallest entity of a programming language, and an application can be broken down to the level of statements for analyzing the relation between them. This granularity provides the highest degree of freedom to the analysis, but it is obviously too fine-grained for parallelization. For example, Figure 7.1a shows a code snippet in which the statement at line 10 performs only one single assignment operation.
Basic Block
A basic block (BB) is defined as a sequence of statements without any control flow in between. BBs are mostly used in traditional compilers as analysis units in which instruction level parallelism is explored. However, for exploring coarse-grained parallelism, they are not well suited. In practice, it is easy to find examples where BBs are either too big or too small. For instance, a basic block is shown in Figure 7.1b from line 13 to 14 in the loop body. It is composed of a sequence of function calls whose execution times can be very long. Since these function calls belong to one BB, they will be seen as a single node in analysis. In such situation, potential parallelism could be overlooked. On the other hand, BBs can also be very simple like the one at line 16 of figure 7.1b. It contains only a single statement with one addition operation. Extracting parallelism from such a BB is impossible.

Function
In compilers, functions are normally subroutines with their own stacks. At this level, only function calls are analyzed and the rest of the code is considered irrelevant for the analysis. Similar to BBs, functions can also be too coarse or too fine-grained depending on the application and on the programming style of the developer. For example, in Figure 7.1c, \( \text{f3} \) and \( \text{f4} \) are two functions. \( \text{f3} \) has a very small function body which indicates that its execution time is short. However, \( \text{f4} \) has a larger body and performs more computations than \( \text{f3} \) does. Due to the difference in the behavior, \( \text{f3} \) and \( \text{f4} \) shall not be equally treated in the partitioning process. For parallelization, solely using function as granularity is therefore not enough.

From the above discussion, it can be seen that statement, BB and function are not well suited to be used as the granularity level for extracting coarse-grained parallelism. This is mainly caused by the fact that these constructs are defined from a control oriented pointed view, with which the amount of computations and the internal connection within each construct are both not taken into account. For the search of coarse-grained parallel tasks, a new granularity is needed.

In practice, the control structure of parallel tasks can be various. They can be constructed from a simple sequence of statements as well as complex control flows such as loops, if-then-else, etc. For this reason, it is required that the new granularity concept is general enough to support analyzing code with different control structures. Besides, the data dependence should also be
taken into account in order to increase the data locality within the potential parallel tasks. These are addressed in the MAPS partitioning tool with a new concept.

### 7.3 Coupled Block

This section presents details about the Coupled Block (CB). First, the CB definition is given; then the design criteria of the CB are discussed.

#### 7.3.1 CB Definition

Remember that in WSCDFGs nodes are IR statements. They are the most fine-grained elements that can be used by analyses to represent a program and provide the highest flexibility for defining a subpart of the program. Therefore, the definition of CB is based on the WSCDFG.

**Definition (Coupled Block).** A coupled block (CB) \( G' = (S', E'_c, E'_d, W'_s, W'_c, W'_d) \), is a subgraph of a WSCDFG \( G = (S, E_c, E_d, W_s, W_c, W_d) \), where:

- \( S' \subseteq S \) is the node set;
- \( E'_c \subseteq E_c \) is the control edge set;
- \( E'_d \subseteq E_d \) is the data edge set;
- \( W'_s(s) = W_s(s) \) is the weight of node \( s \);
- \( W'_c(e_c) = W_c(e_c) \) is the weight of control edge \( e_c \);
- \( W'_d(e_d) = W_d(e_d) \) is the weight of data edge \( e_d \);
- \( \exists! s_{entry} \in S' : Pred(s_{entry}) \cap S' = \phi \land \forall s_i \in S', s_{entry} \text{ dom } s_i \);  
  \( s_{entry} \text{ dom } s_i \) means \( s_{entry} \) dominates \( s_i \) \cite{91}, i.e. all execution paths to \( s_i \) include \( s_{entry} \);
- \( \exists! s_{exit} \in S' : Succ(s_{exit}) \cap S' = \phi \land \forall s_i \in S', s_{exit} \text{ pdom } s_i \);  
  \( s_{exit} \text{ pdom } s_i \) means \( s_{exit} \) postdominates \( s_i \) \cite{91}, i.e. all execution paths from \( s_i \) include \( s_{exit} \);
- \( \forall s_i \in S : (s_{entry} \text{ dom } s_i \land s_{exit} \text{ pdom } s_i) \rightarrow s_i \in S' \);

Altogether, the above three conditions ensure that a CB has at most one entry and one exit;

- \( W_{ij}(s_i, s_j) = \sum_{e'_d(s_i, s_j) \in E'_d} W'_d'(e'_d) + \sum_{e'_c(s_i, s_j) \in E'_c} W'_c'(e'_c) \);

  \( W_{ij}(s_i, s_j) \) is the overall weight of the edges between \( s_i \) and \( s_j \); and

  \[ \sum_{s_i, s_j \in S} W_{ij}(s_i, s_j) \frac{1}{|S'|} \geq T, \text{ } T \text{ is a user given parameter; } \]

  *I.e. the ratio between the sum of all edge weights and the number of nodes \(|S'|\) must be larger than a user given parameter \( T \).*

In other words, a CB is a single-entry/single-exit subgraph of a WSCDFG, whose nodes are strongly connected by control/data dependencies. The definition is developed with respect to mainly two criteria, namely schedulability and data locality. In the following sections, the two criteria will be discussed in detail.
### 7.3.2 Schedulability Constraint

The ultimate goal of the partitioning process is to parallelize a sequential application into smaller parallel tasks, and each of these tasks finishes part of the functionality of the application. From this point of view, these tasks are originated from parts of the sequential implementation, which are **Code Blocks** in the original source code. Generally speaking, a code block can contain any arbitrary code. Figure 7.2 shows some example code blocks which can be constructed from a piece of C code, in which a) shows a code block with one line of code, b) is an example containing a complete loop structure, and the code block in c) includes the condition of a loop and part of the loop body.

![General Code Block Example](image)

Although the construction of code blocks is very flexible, not every code block can be a good candidate for parallel tasks. For instance, the code block in Figure 7.2c contains a complete complex control structure which is part of a loop. It does not have a deterministic exit point, and partitioning the code according to it would imply that the task following it must have multiple entries. However, the execution of a task can only begin with one starting point, otherwise the task scheduler would not be able to know where to start the execution. Therefore, if parallel tasks need to be constructed from the given code block, the code would need to be massively restructured, and this could be very complex.

For this reason, schedulability is the first criterion for the construction of CBs. Specifically, this requires that each CB must have a unique entry and exit, from which the execution of the CB can be started and ended. In the CB definition, the dominance and postdominance relations are used to ensure the fulfillment of this constraint. Note that, since dominators and postdominators have been well studied in compiler literature such as [104], they will not be further discussed in this thesis.

Compared to the definition of the BB which does not allow branches inside a BB, the schedulability constraint is more relaxed concerning the internal control flow. Through this, CBs can be constructed from a wider range of source code than BBs. For instance, the examples in Figure 7.2 a) and b) have completely different internal control flows, but they both fulfill the constraint.

### 7.3.3 Data Locality Constraint

Generally speaking, data locality refers to the phenomenon that a piece of code frequently accesses the same variable or memory location. For MPSoC programming, improving the data locality of
7.3. COUPLED BLOCK

a parallel task means that a small amount of data needs to be transferred from other tasks. Since communicating data from a processor to another processor is very expensive in terms of the required transfer time, parallel tasks with high data locality normally have less communication overhead than those with low data locality.

Typically, data locality can be observed through data flow graphs. Figure 7.3 shows some example data flow graphs, in which solid arrows represent control flow and dashed arrows represent data flow. In example a), since \( s_i \) and \( s_j \) have no data dependence, they do not share any data. In comparison, \( s_m \) in example b) writes variable \( a \) which is later read by \( s_n \), they shared the same variable \( a \). The weight of the data edge between \( s_m \) and \( s_n \) is 4, when \( a \) is a 4-byte integer. In c), there is a backward edge which shows the situation when the statements \( s_p \) and \( s_q \) belong to a loop. Assumes that the loop is iterated 100 times in the execution, then the edge weight will be 400, which shows a strong connection between the two statements.

For constructing CBs, putting \( s_i \) and \( s_j \) of the a) example in one CB does not increase data locality, because no data is shared between them. On the other hand, since the variable \( a \) is shared by \( s_p \) and \( s_q \) multiple times, it makes sense to create a CB including the two statements to increase data locality. Otherwise, the variable might need to be frequently communicated between parallel tasks, which is not desired.

Therefore, the second criterion of the CB definition is data locality, which is controlled by the ratio between the weight of all edges inside a CB and the number of nodes in the CB. Note that a user given parameter \( T \) is used in the definition to manage how strong the nodes should be connected. The higher the \( T \) is, the stronger the nodes of CB are connected.

![Figure 7.3: Data Locality Examples](image)

![Figure 7.4: Data Locality Example](image)
Two examples are shown in Figure 7.4 with the values of the edge weight labeled beside each edge, and the node weights are assumed to be the same in both graphs. Since the graphs are very simple, it can be directly seen that the nodes in the example b) are more tightly connected through edges in comparison to the nodes in the example a). When this phenomenon is connected with data dependence, then it can be said that a parallel task which is created from the example b) has a higher data locality than the one created from the example a).

7.4 CB Generation and WSCDFG Partitioning

For creating CBs in WSCDFGs, a heuristic algorithm, Constrained Agglomerative Hierarchical Clustering (CAHC), is developed and used in the MAPS framework. This section first gives the definition of a partitioned WSCDFG. Then the condition of an optimum partition which is the ultimate goal of the CB generation, is introduced. Afterwards, detail about the CAHC algorithm is presented. Finally, the concept of iterative clustering is explained.

7.4.1 WSCDFG Partition

Since CBs are subgraphs of WSCDFG, a partitioned WSCDFG can be seen as a graph whose nodes is a mixture of IR statements and CBs. Below is the definition of a WSCDFG partition:

7.2. Definition (WSCDFG Partition). Given a WSCDFG $G = (S, E_c, E_d, W_s, W_c, W_d)$, its partition $P = (S', CB, E'_c, E'_d, W'_s, W'_c, W'_d)$ is a graph with the following properties:

- $S' \subseteq S$ is the set of statements which do not belong to any CB;
- $CB$ is the set of coupled blocks in $G$;
- $E'_c \subseteq (S' \times S') \cup (S' \times CB) \cup (CB \times CB) \cup (CB \times S')$ is the set of control edges;
- $E'_d \subseteq (S' \times S') \cup (S' \times CB) \cup (CB \times CB) \cup (CB \times S')$ is the set of data edges;
- $W'_s$ is the node weight;
- $W'_c$ is the control edge weight;
- $W'_d$ is the data edge weight;
- $\forall cb \in CB : Nodes(cb) \cap S' = \phi$,
  $Nodes(cb)$ is the set of nodes in $cb$ and the statements in $S'$ do not belong to any CB;
- $\forall cb_i, cb_j \in CB : Nodes(cb_i) \cap Nodes(cb_j) = \phi$, i.e. CBs cannot have common nodes;
- $\{Nodes(cb) : cb \in CB\} \cup S' = S$;
  I.e. the union of all nodes in CBs and the statements not belonging to any CB equals the nodes in the WSCDFG $G$.
- $\forall e_c(s_i, s_j) \in E_c : (s_i \in cb \land s_j \notin cb) \Rightarrow \exists e'_c(cb, s_j) \in E'_c, W'_c(e'_c) = W_c(e_c)$;
- $\forall e_c(s_i, s_j) \in E_c : (s_i \notin cb \land s_j \in cb) \Rightarrow \exists e'_c(s_i, cb) \in E'_c, W'_c(e'_c) = W_c(e_c)$;

-
There are 5 definitions:

1. \( s \in S \)
2. \( E \in E \)
3. \( W \in W \)
4. \( C \in C \)
5. \( P \in P \)

**Definition 1:**
\[
\forall e(d(s_i, s_j) \in E_d) \land (s_i \in cb \land s_j \notin cb) \Rightarrow \exists e'(d(c, s_j) \in E_d) \land W'(e') = W(d) \\
\]

**Definition 2:**
\[
\forall e(d(s_i, s_j) \in E_d) \land (s_i \notin cb \land s_j \in cb) \Rightarrow \exists e'(d(s_i, cb) \in E_d) \land W'(e') = W(d) \\
\]

**Definition 3:**
\[
\forall e(d(s_i, s_j) \in E_d) \land (s_i \in cb_m \land s_j \in cb_n) \Rightarrow \exists e'(d(c_m, c_n) \in E_d) \land W'(e') = W(d) \\
\]

**Definition 4:**
\[
\forall e(d(s_i, s_j) \in E_d) \land (s_i \in S \land s_j \in S) \Rightarrow \exists e'(d(s_i, s_j) \in E_d) \land W'(e') = W(d) \\
\]

**Definition 5:**
\[
\forall e(d(s_i, s_j) \in E_d) \land (s_i \in S \land s_j \in S) \Rightarrow \exists e'(d(s_i, s_j) \in E_d) \land W'(e') = W(d) \\
\]

**Definition 6:**
\[
\forall e(d(s_i, s_j) \in E_d) \land (s_i \in S \land s_j \in S) \Rightarrow \exists e'(d(s_i, s_j) \in E_d) \land W'(e') = W(d) \\
\]

The last two definitions ensure that the edges between the statements which do not belong to any CB are kept in the partition.

A simple WSCDFG example is shown in Figure 7.5 a) with a partitioned version presented in b). Note that, since \( s_3 \) and \( s_4 \) are merged into \( cb_1 \), the related data edges, \( e_d(s_1, s_3) \) and \( e_d(s_2, s_4) \), end at the CB in the partitioned WSCDFG.

**Figure 7.5: WSCDFG Partition**

An unpartitioned WSCDFG can be seen as a special partition with \( CB = \phi \). Theoretically, given a WSCDFG, a huge number of possible partitions can be derived from it, because the CB creation is very flexible. Hence, it is necessary to define the optimality of the partition in order to guide the generation of CBs.

### 7.4.2 Partition Optimality

It is difficult to define the optimality of a partition, when only one CB is considered, because it only represents a part of the partition. A global measure is required to evaluate whether a partition is good or not. For this, some definitions need to be introduced first.
7.3. Definition (Node-Node Similarity). Given a WSCDFG, $G = (S, E_c, E_d, W_s, W_c, W_d)$, the similarity between any two nodes in the graph, $\text{sim}(s_i, s_j), s_i, s_j \in S$, is defined as:

- $\text{sim}(s_i, s_j) = \frac{W_{ij}(s_i, s_j)}{\text{Dist}(s_i, s_j)}$

$\text{Dist}(s_i, s_j)$ is the distance between $s_i$ and $s_j$, i.e., the length of the shortest control path between the nodes; and

$W_{ij}(s_i, s_j)$ is the overall weight of the edges between $s_i$ and $s_j$, as defined in Section 7.3.1.

Similarity is a term commonly used in solving graph clustering problems. Typically, it shows how close two nodes are. In the node-node similarity definition, the distance between two nodes only takes into account the control edges. Given two nodes, the shorter the distance between them is, the more similar they are. Figure 7.6 shows two examples in which control edges are drawn with solid lines. It can be seen from the figure that $s_1$ and $s_3$ in the example b) are separated from each other by another node, and therefore they are less similar than the $s_1$ and $s_2$ in the example a) are.

In graph clustering, the similarity factor can be used to prevent a cluster from spanning a large portion of the graph. Here, it is used to denote the closeness of any two nodes in a WSCDFG.

![Node-Node Similarity Examples](image)

Figure 7.6: Node-Node Similarity Examples

7.4. Definition (CB Internal Similarity). Given a CB, $CB' = (S, E_c', E_d', W_s', W_c', W_d')$, its internal similarity is defined as:

- $\text{iSim}(CB') = \frac{\sum_{s_i, s_j \in S'} \text{sim}(s_i, s_j)}{|S'|}$

The CB Internal Similarity describes the average similarity for all the node pairs which can be found in a CB. Note that it is divided by the cardinality of the set $S'$. By doing this, a CB with heavier connections to few nodes will be have a higher similarity than the one with light connections to many nodes, which implies that higher data locality can be achieved than the former.

When the optimality of a parallel application is considered, it is natural that the speedup in comparison to the sequential version is used as the target function. However, here, in the partitioning phase, the speedup number is not available, because the parallel version is yet to be developed. Therefore, data locality is used as a replacement measure to define the optimality of partitions, since higher data locality can lead to less inter-processor communication and hence better parallel performance. A WSCDFG partition is seen as optimal when the highest overall data locality is achieved. Given the node and the CB similarity, the optimality of a WSCDFG partition can be defined as follows:
7.5. **Definition (Optimum WSCDFG Partition).** Given the set of all possible partitions of a given WSCDFG $G$, $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$, a partition $P^* \in \mathcal{P}$ is said to be optimal if:

\[
P^* = \arg\max_{p \in \mathcal{P}} \left( \sum_{cb \in CB(p)} (isim(cb)) \right)
\]

$CB(p)$ is the set of couple blocks in a partition $p$.

Finding the optimal partition can be seen as a graph clustering problem which has been proven to be NP-hard even for simple algorithms such as $k$-means with euclidean distances [116]. For this reason, MAPS employs a heuristic algorithm to partition WSCDFGs.

### 7.4.3 CAHC Algorithm

From the graph theory’s point of view, CBs can be seen as node clusters which are strongly connected by control/data edges. The problem of finding such clusters in a huge graph also exists in the data mining area, where clusters of data are searched for according to given criteria. There are algorithms developed for this purpose, and MAPS chooses to extend an existing work for partitioning.

The Constrained Agglomerative Hierarchical Clustering (CAHC) algorithm used in MAPS is a modified version of the Density Boundary SCAN (DBSCAN) [117] algorithm. The algorithm converts a graph clustering problem into a vector clustering problem by using the similarity measures introduced in the previous section. The execution of the algorithm can be roughly separated into three steps:

- **Dense Core Calculation**
  Dense cores are node sets whose members are tightly connected with edges. They are first calculated as skeletons of CBs.

- **CB Generation**
  Since a dense core only includes the strongly connected nodes, it does not directly fulfill the CB definition. An additional step is used here to create CBs from dense cores.

- **Iterative Clustering**
  In order to obtain CBs on different granularity levels, fine-grained CBs are further clustered into coarse-grained ones in the final step of the CAHC algorithm.

The CAHC algorithm mainly uses the DBSCAN algorithm in the first step to find clusters of nodes which are tightly connected. Though, some parameters are changed in order to employ the control and data information.

**Dense Core Calculation**

The goal of dense core calculation is to find out those nodes which are tightly connected by dependencies. Since the number of nodes in a WSCDFG can be numerous, it is necessary to find the nodes which are strongly connected with their neighbors (predecessors and successors). For this, the concept of core-node is used, which is derived from the core-condition in the DBSCAN algorithm.
7.6. Definition (Core-Node). A core-node \( c \) in a WSCDFG, is a node whose weighted degree, \( wDeg(c) \), fulfills the following condition:

\[
    wDeg(c) = \frac{\sum_{s_i \in \text{Pred}(c) \cup \text{Succ}(c)} \text{sim}(s_i, c)}{|\text{Succ}(c) \cup \text{Pred}(c)|} \geq m
\]

The condition \( wDeg(c) \geq m \) is called the core-condition, and the parameter \( m \) is the average weighted degree of a WSCDFG, which is calculated by the following equation:

\[
    m = \frac{\sum_{s_i \in G} wDeg(s_i)}{|G|} \quad (7.1)
\]

The core-condition here is different from the one used by the DBSCAN algorithm in that it sets a minimum number of neighboring nodes that a core-node shall have. Therefore, a core node can be located in any position in a WSCDFG as long as its connectivity to its neighbor(s) is higher than average. However, a core node represent only one node in a graph and can have multiple predecessors and successors like the example in Figure 7.7 shows, it is necessary to determine which node exactly has a strong connection to a core-node. For this purpose, the concept of \( \epsilon - \text{Neighbor} \) is used.

7.7. Definition (\( \epsilon \)-Neighborhood). The \( \epsilon - \text{neighborhood} \) of a node \( s_i, N_\epsilon(s_i) \), is a set of nodes which fulfills the following condition:

\[
    N_\epsilon(s_i) = \{s_x \mid s_x \in \text{Succ}(s_i) \cup \text{Pred}(s_i), \text{sim}(s_i, s_x) > \epsilon\}
\]

If a node \( s_x \in N_\epsilon(s_i) \), then \( s_x \) is an \( \epsilon - \text{Neighbor} \) of \( s_i \). The parameter \( \epsilon \) is calculated by the following equation:

\[
    \epsilon = \frac{\sum_{s_i, s_j \in G} \text{sim}(s_i, s_j)}{|\{(s_i, s_j) | \text{sim}(s_i, s_j) \neq 0\}|} \quad (7.2)
\]
Note that, when two nodes are not directly connected by a control or data edge, their similarity is zero. Therefore, \( \varepsilon \) is in fact the average similarity between those connected nodes. Figure 7.8 shows an example of \( \varepsilon - \text{Neighbor} \) which illustrates the situation that an \( \varepsilon - \text{Neighbor} \) has an above average connectivity. Based on the definitions of \( \varepsilon - \text{Neighbor} \) and core-node, the relation of Directly Density Reachable can be derived as follows:

7.8. **Definition (Directly Density Reachable).** A node \( s_x \) is said to be Directly Density Reachable (DDR) from a node \( s_i \), w.r.t \( \varepsilon \) and \( m \), if \( s_i \) is a core-node and \( s_x \in N_\varepsilon(s_i) \). This relation is denoted as:

\[ s_x \text{ DDR}_{\varepsilon,m} s_i \]

Using the DDR relation, light-weighted connections to a core-node are excluded. For example in Figure 7.8, assume that \( s_i \) is the only \( \varepsilon - \text{Neighbor} \) of \( s_j \), then \( s_i \text{ DDR}_{\varepsilon,m} s_j \). The concept can be extended to a chain of nodes, which gives the relation of Density Reachable (DR).

7.9. **Definition (Density Reachable).** A node \( s_x \) is said to be Density Reachable (DR) from a node \( s_i \) w.r.t \( \varepsilon \) and \( m \), if:

\[ \exists \{ z_i | i \in \{1, 2, 3...p\} \}, z_1 = s_i, z_p = s_x, \forall i = 1,...,p-1, z_{i+1} \text{ DDR}_{\varepsilon,m} z_i \].

This relation is denoted as \( s_x \text{ DR}_{\varepsilon,m} s_i \)

By applying the DR relation, a sequence of nodes can be strongly connected like the example in Figure 7.9 shows. Since \( s_k \in N_\varepsilon(s_i) \), \( s_i, s_j \) are core-nodes and \( s_i \text{ DDR}_{\varepsilon,m} s_j \), then \( s_k \text{ DR}_{\varepsilon,m} s_j \). Given the definition DDR and DR relations, the dense core is defined as follows:

7.10. **Definition (Dense Core).** In a WSCDFG \( G \), a dense core \( C \) is a set of nodes of \( G \) that satisfies the following conditions:

\[ C \subseteq \text{Nodes}(G) \land C \neq \emptyset; \]
According to this definition, all strongly connected nodes in a WSCDFG are clustered into a set of dense cores. For instance, the nodes \( s_i, s_j, s_k \) in Figure 7.9 would form a dense core. These nodes are the skeleton of CBs. However, dense cores are not CBs yet. Based on the example of Figure 7.9, Figure 7.10 illustrates the subgraph that a dense core could cover. As it can be seen from the diagram that there are multiple control edges entering and exiting the subgraph, which violates the schedulability constraint of the CB definition. Therefore, a post-processing is needed to finally build CBs from the dense cores, which is the second part of the CAHC algorithm, CB generation.
CB Generation

According to the CB definition, two constraints need to be met in order to construct CBs from dense cores, schedulability and data locality. To find the set of nodes which can fulfill the schedulability constraint, the CAHC algorithm uses the concept of induced set.

7.11. Definition (Induced Set). Given a set of nodes $C$ in a WSCDFG $G$, its induced set $IS(C)$ is the smallest node set which fulfills the following conditions:

- $C \subseteq IS(C) \subseteq \text{Nodes}(G)$;
- $\exists s_{\text{entry}} \in IS(C) : \text{Pred}(s_{\text{entry}}) \notin IS(C) \land \forall s_i \in IS(C), s_{\text{entry}} \text{ dom } s_i$;
- $\exists s_{\text{exit}} \in IS(C) : \text{Succ}(s_{\text{exit}}) \notin IS(C) \land \forall s_i \in IS(C), s_{\text{exit}} \text{ pdom } s_i$;
- $\forall s_i \in \text{Nodes}(G) : (s_{\text{entry}} \text{ dom } s_i \land s_{\text{exit}} \text{ pdom } s_i) \Rightarrow s_i \in IS(C)$.

Note that the last three conditions are similar to the CB schedulability definition. In fact, if a subgraph is constructed from an induced set, then the conditions will become the same as those of the CB in the context of the subgraph. Figure 7.11 gives an example induced set which is constructed from the dense core in Figure 7.10. It can be seen that some neighboring nodes are included into the induced set in order to fulfill the conditions. In the example, the dense core nodes $s_k$ and $s_j$ happen to be entry and exit. However, this is not necessarily the case for the induced set of all dense cores. It is often the case that the entry and exit nodes do not have a strong connection to other nodes.

The induced set guarantees that if a CB is constructed from a set of nodes, it can be scheduled. To meet the data locality constraint, the internal connectivity of the nodes needs to be checked using the equation given in the CB definition. Nevertheless, since dense cores are created without considering the user given granularity parameter $T$ in the CB definition, it can happen that a dense core is more coarse-grained than the parameter defines. Therefore, directly constructing CBs from dense cores might result in too coarse-grained CBs. To prevent this from happening, the

Figure 7.11: Induced Set Example
construction of CB starts from the induced set of the center of a dense core and gradually includes more nodes until the CB data locality constraint is met. The dense core center is determined using the following definition:

**7.12. Definition (Dense Core Center).** Given a dense core \( C \), its center \( s_c \) is defined as the node with the highest weighted degree, i.e.:

\[
    s_c = \text{argmax}_{s_i \in C} (w\text{Deg}(s_i))
\]

Finally, given a dense core \( C \), the set of nodes \( K \) which should belong to a CB, is determined using the following criteria:

- \( K = IS(C^*) \), \( C^* \subseteq C \); and
- \( \sum_{s_i,s_j \in IS(C^*)} W_{ij}(s_i,s_j) > T \).

\( C^* \) is a subset of \( C \) and initially contains only the dense core center \( s_c \) of \( C \). When \( C^* \) is extended, the rest nodes in \( C \) are added one-by-one according to their weighted degree. The node with a heavier degree is always added first. Thereby, the search for \( K \) given a dense core \( C \) can be done without trying all possible subsets of \( C \) and can therefore be finished within a short period of time.

After the creation of CBs, a WSCDFG partition can be easily constructed by removing those control/data edges between nodes within CBs and adding new ones for the connection between CBs and non-clustered IR statements. This process is very straightforward.

### Iterative Clustering

The hierarchical clustering part of the CAHC algorithm is implemented based on the fact that a WSCDFG partition can be further clustered into a more coarse-grained partition. Given a WSCDFG partition \( P = (S, CB, E_c, E_d, W_s, W_c, W_d) \), when IR statements and CBs are equally considered as graph nodes, all definitions like node-node similarity and dense core introduced above can be applied on the members in \( S \) and \( CB \) again. Thereby, a more coarse-grained partition \( P^n = (S^n, CB^n, E^n_c, E^n_d, W^n_s, W^n_c, W^n_d) \) can be obtained from a partition \( P^{n-1} = (S^{n-1}, CB^{n-1}, E^{n-1}_c, E^{n-1}_d, W^{n-1}_s, W^{n-1}_c, W^{n-1}_d) \) which is resulted from an earlier iteration of the CB generation procedure.

Remember that a WSCDFG \( G \) can be seen as a special partition \( P^0 \) with its \( CB = \phi \), then using the clustering process described above iteratively on \( P^0 \) multiple times can result in a set of partitions, \( P^0 \Rightarrow P^1 \Rightarrow \cdots \Rightarrow P^n \). This iterative clustering process is illustrated in Figure 7.12, in which it can be seen that after each iteration some nodes are clustered into more coarse-grained ones.

Between different iterations of the clustering process, the CB parameter \( T \) is not changed. It controls the CB granularity within each iteration. After the clustering for a WSCDFG is finished, i.e. no new CB can be found, a set of partitions are produced at different granularity levels. Since a WSCDFG represents only one function in an application, to decide which partition or granularity is best suited for the whole application, it is necessary to analyze all functions globally. The next section explains the global analysis in the MAPS partitioning tool, which is implemented for this purpose.
7.5 Global Analysis

The CB generation procedure described in the previous section does not directly determine whether the granularity selected for a given function matches the granularity suitable for the other functions in the application. In order to cope with this problem, a global analysis is performed in the MAPS partitioning tool based on the partitions produced by the CAHC algorithm.

The basic idea of the analysis is to balance the “size” of CBs. The overall node weight of the CB is used here as the measure of “size”. Remember that the weight of a node in the WSCDFG is its estimated execution cost, which is related to the execution time. Having CBs with a similar “size” can help to avoid the case that a parallel task runs for a long time but another runs just for a short period. In such situation, the processing element running the short task would need to wait for the one running the long task to finish, which is not desirable.

According to the above idea, given a set of partitions for each function in the target application, the most suitable partition is determined in MAPS with the following steps:

- First, the application call graph is traversed in a bottom-up fashion (finishing at the main function) and a CB statistic is made with respect to the iteration number of the partition and the source function.

- Second, the CB weights are compared against each other in order to find a common granularity for different functions. To make the decision, a simple load balancing is performed following the first fit algorithm introduced in [118]. With this value, the granularity of the final parallel tasks is equalized.

- Finally, for each function, a partition is selected, whose CBs weights are the closest to the common granularity value.

Alternatively, the global analysis procedure can also be turned off. In this case, the user needs to manually decide which functions in the application should be partitioned and how many iterations each function should be partitioned. Figure 7.13 shows the GUI interface of the MAPS IDE, with which the user can semi-automatically partition an application. For convenience, functions are sorted according to their execution costs. The user needs to just check the functions to be partitioned and specify the iteration numbers. Such a semi-automatic partitioning procedure is
supported by MAPS in parallel to the automatic global analysis, so that the user can have more freedom in deciding the parallelization strategy.

Figure 7.13: Semi-Automatic Partitioning Control

7.6 User Refinement

In the MAPS tool flow, the CBs produced by the CAHC algorithm and the global analysis are directly considered by the tool as parallel tasks. Nevertheless, before parallel code is generated from them, the user is given the chance to refine the results. Since the automatic partitioning procedure described in the previous sections analyzes the application mainly based on its internal control/data dependence information, application knowledge such as the structure of a video frame are not taken into account. Such information is typically very specific to an application and difficult to be extracted automatically. Therefore, the user refinement step is necessary in the MAPS framework to allow the programmer to inspect the result of the automatic tool and perform modifications when necessary. Through this, the programmer can use his knowledge in the application domain to further improve the parallelization result.

The user refinement is supported through the graphical user interface (GUI) of the MAPS IDE. Figure 7.14 is a screen shot of the corresponding GUI. On the left-hand side, the application source code is displayed in the MAPS C code editor, which highlights the code regions of different tasks with different background colors. The programmer can directly see which lines of source code are clustered together. On the right hand side, a graphical view of the corresponding WSCDFG partition is given, through which the programmer can see the IR statements in CBs and the dependences between them. For example, the topmost CB node shown in the screen shot contains the IR statements for two function calls `def1` and `use1`, and the node corresponds to the task covering the source lines 114, 115. The control and data edges are annotated in the graph with their weights, and the connectivity between nodes is then directly visible to the user. By using the MAPS IDE, the programmer can easily modify the partitioning results by removing/modifying the existing tasks or adding new tasks.
7.7 Summary

This chapter introduces the partitioning process in the MAPS development flow. A granularity concept, coupled block, is used in the partitioning tool to capture different granularity levels. In order to find parallelism in an application automatically, the tool uses the CAHC algorithm and a global analysis process. The generated tasks are visualized by the MAPS IDE and the programmer is given the chance to change the result when necessary. From a programmer’s perspective, the partitioning process is completely under his control.
Chapter 8

High-Level Simulation

In the MAPS framework, high-level simulation is employed for early MPSoC software development. The meaning of early here is two-fold. First, when the MPSoC design is in a very early stage and no VP or hardware prototype is available, programmers can use the high-level simulation facility of MAPS to execute and test MPSoC applications. Second, even when a hardware prototype is already there, since parallel applications are difficult to debug, they can be first tested using high-level simulation before being executed on the target platform. Moreover, in addition to the functional test of MPSoC applications, programmers can also get early performance estimates through high-level simulation.

The above mentioned goals are supported in the MAPS framework by the MAPS Virtual Platform (MVP). The MVP consists of two parts, an abstract MPSoC simulator and a special software tool chain. This chapter introduces the high-level simulation approach implemented in MAPS and gives details about the MVP.

The rest of this chapter is organized as follows: as next, Section 8.1 gives a survey on some related works; then, an overview of the MVP is shown in Section 8.2; the details of the MVP simulator is introduced in Section 8.3, and the MVP specific software tool chain is discussed in Section 8.4. Finally, Section 8.6 summarizes the whole chapter.

8.1 Related Work

High-level simulation is a key component in Electronic System Level (ESL) design flows. A lot of MPSoC design frameworks have been developed in the past few years, which use high-level simulation for early design space exploration. Although they all simulate MPSoC at a high abstraction level, different approaches are used to create high-level application models.

Some works use SystemC [119] to model MPSoC applications, e.g. [120] and [121]. In this case, the application behavior must be modeled within SystemC specific C++ classes to get accommodated in the simulation environment. Hence, the source code cannot be directly reused for the target MPSoC, and special treatments are required in these approaches to convert high-level application models to C code for target execution. Typically, this process is called software synthesis.
The framework introduced in [122] also uses SystemC as its high-level modeling language, but it uses a different approach to get the software running on the target. Instead of synthesizing C code from a high-level application model, it has a special runtime implementing the SystemC environment in the target platform. To generate a target executable binary, the programmer only needs to cross-compile the source code which is written in SystemC, and link the output objects with the special runtime library. Although the application model does not need to be changed in the whole process, a SystemC runtime environment must be implemented for the target platform, which is not trivial.

Except for SystemC, other modeling languages are also used for creating high-level models. For instance, Simulink [123] is used in the work which is introduced in [124]. In [125] the author uses UML [126] to describe software behavior. Since these languages cannot be directly compiled into executables, extra code generators are needed to generate C source for the target platform. Besides, such modeling languages are very different from traditional sequential programming languages such as C or C++, therefore, it is difficult to reuse legacy C code. This implies that even if the C/C++ code of an old application is available, the application needs to be modeled from scratch again in a new modeling language.

Some works use C/C++ as high-level modeling language, e.g. [127], [128] and [57], because C is until today the most used programming language in the embedded domain [77]. Nevertheless, the target application must be modeled as Kahn Process Networks (KPNs) [47] by using special API, which restricts the flexibility of high-level modeling.

Modeling applications is just one aspect of high-level simulation, another is abstracting the underlying MPSoC architecture. In this area, a lot of works have been done which are focused on the modeling of OS. For example, [129] introduces a simulation framework which uses an abstract RTOS model for task scheduling. However, since software behavior is not included in its application model, the framework is more suitable for the overall system design than the software development.

Some frameworks model OS together with software behavior at a high abstraction level by using SystemC ([130] and [131]) or SpecC ([132] and [130]) as their modeling language. Such approaches normally require the user to insert timing annotations into the source code explicitly in order to let software consume execution time in high-level simulation.

Instruction set simulation is a typical source of obtaining the required timing information. For example, the Sesame [133] and MPA [70] frameworks determine software delays by first running the target application in an instruction-set simulator (ISS) and then back-annotating the result to the corresponding high-level models. Different from the MPA, the Sesame framework additionally supports the use of ISS when its high-level simulation is running. The timing information is extracted and back-annotated on-line at runtime. A similar approach is also implemented in the work introduced in [134] to automate the annotation process.

Since the simulation speed of ISS is limited, timing annotated native execution has been proposed, which is typically supported by a tool capable of extracting timing information automatically. For example, the work introduced in [135] statically extracts timing information from target binaries which are cross-compiled from intermediate C source code. The results are thereafter annotated back to the natively compiled code. In comparison, the approach taken in [136] is more direct. It modifies the backend of the GCC compiler to produce timing annotated SystemC code for native simulation.

The high-level simulation tool of the MAPS framework, the MAPS Virtual Platform (MVP), is
similar to the VPU work which is introduced in [137] and [138]. The VPU is developed as a part of the CoWare Platform Architect (PA) [139] which is essentially a SystemC modeling framework, and targets both MPSoC hardware architecture exploration.

The design of the MVP solely targets MPSoC software development. With this goal in mind, the approach taken by the MVP differs from the above mentioned frameworks mainly in the following aspects:

- **MPSoC model**
  Since the MVP targets software development, especially functional test, some hardware details are completely simplified in the MVP, e.g. communication architecture. In comparison, frameworks targeting architecture exploration are capable of modeling the complete network-on-chip of an MPSoC, which can be very detailed.

- **A priori knowledge**
  To use the MVP, it is not required that a programmer needs to have deep knowledge of hardware architecture or SystemC modeling experience. Some frameworks are designed for creating platform models with SystemC, and hence the user must learn SystemC before he can use the tools.

- **User interface**
  To create an abstract MPSoC model using the MVP, the programmer can use just mouse-click and drag-and-drop operations through a graphical user interface and finish the modeling process within minutes. For a similar MPSoC platform, longer time is required by tools oriented to SystemC modeling, because the user has much more platform details to take care of.

- **Application model**
  C is the only programming language the programmer needs to know in order to create an application model to run in the MVP. Most other frameworks require that the application source code must be wrapped in SystemC specific C++ classes.

- **Inter-task communication**
  The MVP supports the inter-task communication in parallel applications simply through shared memories which can be accessed by normal C pointer dereferences. This is enough for developing functional correct software. Other frameworks require that the communication between tasks must be explicitly coded as SystemC channel accesses so as to generate traffic on the interconnection network, which is necessary for architecture exploration but not for functional simulation.

- **Application/simulator decoupling**
  In the MAPS framework, the application binaries are separated from the MVP simulator. They are dynamically loaded into the simulator before the simulation starts. To change the applications running in the MVP, the programmer only needs to load a different set of applications. Most other frameworks, on the other hand, compile application source code together into one simulation executable. Each time there is a change in the MPSoC configuration such as the task-to-processor mapping is modified, the simulation needs to be recompiled.

- **Runtime configuration**
  When the simulation is running, the MVP simulator allows its user to dynamically change
some configurations such as the processing element execution speed, task-to-processor mapping, etc. In other frameworks, such modification would require the simulation to be stopped, reconfigured and sometime recompiled to see the effect of the change.

8.2 MVP Overview

The goal of the MVP is to support MPSoC programming with high-level simulation. Therefore, abstract simulation of MPSoC is the main feature of the MVP. To achieve this, SystemC is used as the simulation backend, which is the most used approach for system- and high-level exploration. However, since SystemC is based on C++ and its predefined language constructs were originally designed for hardware simulation, C applications cannot be directly simulated in it. Normally, developers need to manually wrap a piece of C code into SystemC modules in order to run it in SystemC, which can be a tedious process and requires a priori knowledge of SystemC. As the target users of MAPS being software developers who might not be familiar with SystemC or hardware architecture, a special software tool chain is included in the MVP, which allows the MAPS user to develop application directly in C without worrying about the underlying simulation framework.

Figure 8.1 shows an overview of the MVP. Roughly, the MVP can be divided into two parts:

- **Abstract MPSoC simulator**, which is responsible for the high-level simulation of MPSoC architectures; and
- **Software tool chain**, whose final products are native binaries with application behavior embedded and can be loaded into the simulator dynamically.

Figure 8.1: MVP Overview

To use the simulator, a GUI is provided by the MVP as a centralized cockpit for the configuration and control of the simulation process. A programmer can use the GUI to load applications, instantiate processing elements, set processor speed and map applications to processors. The creation of an abstract high-level MPSoC model and its configuration can be done in the GUI completely by mouse-click and drag-and-drop operations within minutes. Additionally, the simulator can also be
configured through files, using which the programmer can easily manage the simulation of different MPSoC configurations.

Moreover, as it can be seen in the figure that the application native binaries are decoupled from the MVP simulator. Here, the meaning of the decoupling is twofold. First, the applications are compiled into shared libraries which are physically independent and separate from the simulator. Second, each application in the MVP is a standalone program in C, which has its own main function and does not interfere with the one of the SystemC simulation kernel. Because of this decoupling, the MVP simulator can be used in a very flexible way.

It is often that an MPSoC programmer needs to simulate different application scenarios each of which needs to run a different set of applications. Using the MVP, the programmer only needs to load a different set of applications in order to simulate these scenarios, which can be easily done through the GUI or the configuration file.

In the following sections, the above mentioned parts of the MVP and their usage will be discussed in detail.

8.3 MVP Simulator

The MVP simulator provides application developers a high-level abstraction of the underlying MPSoC platform. It allows programmers to easily model an MPSoC architecture without taking care of its low-level details, and afterwards simulates the architecture at a high abstraction level.

From a high-level perspective, an MPSoC roughly consists of:

- **Processing elements**, which perform computations;
- **Interconnections**, through which processing elements transfer data between each other;
- **Peripherals**, which are responsible for the interaction with users and external devices.

In the MVP simulator, these components are abstracted with high-level models. Processing elements are abstracted by the so-called Virtual Processing Elements (VPEs). The interconnections for inter-processor communication are modeled using generic shared memories. A virtual I/O peripheral is provided for the display of graphical and text information.

The parallel operation of the components is done by using SystemC as simulation environment. The implementation of the simulator is in line with the loosely-timed coding style which are suggested by the TLM 2.0 standard [140] for software development.

8.3.1 Virtual Processing Element

In the MVP simulator, a VPE is a high-level abstraction of a processing element and the operating system running on it. It is responsible for the control of the execution of the software tasks mapped to it. This includes the decisions of: which task should be executed (i.e. scheduling), how long the selected task should run (i.e. time slicing), and how many operations the task should perform in the given time slot (i.e. execution speed). Conversely, tasks can also request the VPE for OS services like sleep for n microseconds etc. The interaction between a VPE and tasks can be
roughly illustrated as in Figure 8.2, where the VPE sends out an event TASK_RUN to let Task1 execute 200,000 operations for 1ms, and the task requests to sleep for 50ms during its execution.

Technically, tasks and VPEs are implemented as SystemC modules. Their interaction is realized by the events which are communicated through the TLM channels between them. The following paragraphs will introduce the semantics of the VPE model and the task model.

VPE Model

A VPE can be described here as an event-driven finite state automaton which is a 7-tuple $V = (S, s_0, U, O, I, \omega, \nu)$ with:

- $S = \{RESET, RUN, SWITCH, IDLE\}$ is the set of explicit states;
- $s_0 = RESET$ is the initial state of the VPE;
- $U$ is the set of internal variables which represent the implicit states like $TIME\_SLICE\_LENGTH$, etc;
- $O$ is the set of output events for task control, e.g. TASK_RUN;
- $I$ is the set of input events which could be sent by tasks, e.g. TASK_REQ_SLEEP;
- $\omega : S \times I \to O$ is the function, in which functionalities like scheduling are implemented; and
- $\nu : S \times I \to S$ is the next-state function, which manages the OS state.

From a user’s perspective, a VPE simply appears as a parameterized abstract processor. The settings which can be configured by the user are mainly clock frequency, scheduler and task mapping. Presently, three scheduling algorithms are implemented in the VPE, which are:

- **round-robin**: tasks are executed one after another with the same period;
- **earliest deadline first**: the task to be first finished is executed first; and
- **priority based scheduling**: the task with highest priority is executed first.

These parameters can be adjusted both before and during the simulation. This allows the programmer to change the platform and check the application behavior in different scenario conveniently. Besides, the event history can be saved by the simulator in form of Value Change Dump (VCD) files, which can be used to check the VPE activity after the simulation for debugging purposes.
8.3. MVP SIMULATOR

Task Model

A task in the MVP simulator can be seen as an event-driven nondeterministic finite state automaton, which is a 6-tuple $T = (S, s_0, I, O, \omega, \nu)$ consisting of:

- $S = \{READY, RUN, SUSPEND\}$ is the set of task states;
- $s_0 = READY$ is the initial state;
- $I$ is the set of input events, which are sent by the VPE like TASK_RUN;
- $O$ is the set of output events, such as TASK_REQ_SLEEP;
- $\nu : S \times I \to P(S)$ is the next-state function.
  
  Since $\nu$ could return multiple states, the power set of $S$ ($P(S)$, which includes all possible subsets of $S$) is used as the result of $\nu$; and
- $\omega : S \times I \to P(O)$ is the output function.
  
  Similar to $\nu$, the result of $\omega$ is the power set of $O$, $P(O)$.

It can be seen from the above definition that the task model is nondeterministic in that, the next-state function can return a set of possible states. In reality, this corresponds to the case that the status of a task can be changed from RUN to either READY or SUSPEND. The former normally happens when the granted time slice is used up, and the latter can occur when the sleep function is invoked in the code in order to suspend the task for a while.

Figure 8.3 shows the state transition diagrams of the VPE and the task together with an example event which can be exchanged during the simulation. The VPE decides which task should run in the SWITCH state, and then awakes the READY task with a TASK_RUN. Afterwards, the VPE is switched to the RUN state. Correspondingly, upon receiving the TASK_RUN event, the state of the task is immediately switched from READY to RUN.

In this model, the application behavior, which is encapsulated in the user provided shared library, serves as the output and the state transition function of the task and is executed in the RUN state. Since the programmer only provides the C source code, i.e. the functionality of the application, the control of the state machine needs to be inserted. The MVP provides a software tool chain which hides this insertion procedure from the developer and keeps the application C code intact so that it can be reused. Section 8.4 will give more details about this, when the programming support is discussed.

![Figure 8.3: VPE/Task State Transition](image-url)
8.3.2 Generic Shared Memory

Since the MVP is focused on the high-level functional simulation, no complex interconnection is modeled in it. Nonetheless, in order to enable the communication between parallel tasks, the MVP provides so called Generic Shared Memory (GSHM).

From a programmer’s view, the use of the GSHM is similar to the dynamic memory management functions in the C runtime library, except that a string is used to identify each GSHM block so that the communicating tasks can refer to the same block. An example is shown in Figure 8.4. The example shows two tasks which communicate through a GSHM block which is identified through the key string “SHM”. Inside the simulator, all the shared memory blocks are centrally managed by the GSHM manager which keeps a list of them. Since SystemC simulates all modules one after another, i.e., sequentially, no parallel access to the GSHM is possible during the simulation, and all GSHM accesses are thereby automatically synchronized.

The previous example shows a scenario where the data is passed by value, which mostly occurs in multi-processor systems where each parallel task has its own address space and data must be copied between tasks explicitly. Nonetheless, the use of the GSHM can be much more flexible. Since the user provided application binaries are loaded into one simulation process, all tasks running in the MVP simulator implicitly share one common address space. This implies that pointers can also be transferred between tasks without being invalidated, which gives an execution environment similar to a SMP (Symmetric Multiprocessing) machine where the processors share the same address space.

In the MVP, programmers have the freedom of choosing the most suitable communication mechanism for the application. Moreover, this flexibility is also helpful for code partitioning, because the programmer does not have to completely convert the implicit communications to explicit ones in order to test the functionality of the parallelized application. The MVP provides a relaxed test environment, in which partially partitioned prototypes can be simulated and used as intermediate steps towards a cleanly parallelized application. Finally, it needs to be mentioned that the GSHM itself just provides a primitive but easy-to-use way for sharing data between tasks in the MVP. How to transfer the data in target still depends on the implementation of the real platform.
8.3.3 User Interface and Virtual Peripheral

As the MAPS framework focuses on software programming, its user is not likely to be expertized in architecture modeling. Therefore, an easy-to-use interface is required to simplify the creation of MPSoC models and the control of simulation. With this goal in mind, the GUI of the MVP simulator is designed and implemented. Figure 8.5 shows a screen shot of the GUI. It can be used by the MVP user to:

- Create abstract MPSoC models;
- Control the simulation process;
- Get application performance feedback; and
- Interact with the application running in the simulator.

To start creating an abstract MPSoC model, the user only needs to click a button to instantiate the desired number of VPEs, and a VPE control panel will be displayed in the center of the GUI for each VPE. Using the control panel, the user can set the VPE clock frequency and select the scheduling algorithm of the VPE abstract OS. Since the interconnection between processing elements is not modeled in the MVP, no further configuration is needed. The creation of the model can be done within minutes, because of the simple modeling process.

Applications or tasks are loaded into the simulator by using the GUI to select the corresponding native binaries. After the loading succeeds, the names of the application will be displayed in the global task list on the left hand side of the GUI. By dragging an application from the global task list to the VPE task list in an VPE control panel, its mapping to VPE is done, which is very intuitive. The whole configuration of the MVP simulation can be accomplished with the GUI through just mouse operations.

The control of the high-level simulation process is done through the three buttons at the top middle of the GUI window, which control to the start, pause and stop of the simulation. When the simulation is running, the overall VPU usage information is displayed at the bottom of the VPE control panel, the VPE usage of each individual task or application is displayed in the VPE task list directly. Such runtime information provides programmers a rough estimation of the load of processing elements. Base on these information, the programmer can try to optimize the system by experimenting different VPE configurations and task-to-VPE mappings. Since the

![Figure 8.5: MVP User Interface](image)
MVP simulator supports changing task mapping and VPE settings at runtime, such experiments can be done dynamically without stopping the simulation, which saves development time.

Besides, part of the MVP GUI is a virtual display device (on the left of the GUI window in Figure 8.5), through which graphical and textual information can be directly shown without using the host machine. Moreover, the GUI supports transferring keyboard input from it to the application running in the simulator. Thereby, interactive applications can also be simulated in the MVP.

In the host machine, the GUI runs in a separate thread in order to avoid interfering the SystemC thread directly. The communication and the synchronization between them is realized through mutexes and messages in the host machine.

8.4 Software Tool Chain

The MVP software tool chain is mainly responsible for two things:

- First, it provides an execution environment for MPSoC applications. A set of API functions are provided for applications to do inter-VPE communication, scheduling, etc.

- Second, in order to be simulated in the MVP, applications need to be given to the simulator as special dynamic libraries, and the creation of such libraries is supported by the tool chain.

8.4.1 Execution Environment

The programming language supported by the MVP is C. This is mainly due to the fact that C is until today the most used programming language for embedded software development. The difference between running C applications in a real MPSoC platform and in the MVP is illustrated in Figure 8.6. In a typical MPSoC platform, applications run on top of the whole hardware/software hierarchy. As Figure 8.6a shows, below an application, there are middleware such as network protocol stack, operating system such as Linux and MPSoC hardware. However, in the MVP, the operating system and the MPSoC hardware platform are abstracted by the MVP simulator. Between the simulator and the application, there is the MVP API which allows the latter to interact with the former.

![MVP Execution Environment](image)

Figure 8.6: MVP Execution Environment

The MVP API consists of a small number of functions for developers, which enable the communication and the synchronization between tasks, the interaction with the VPE in the simulator, and the access to the virtual peripherals. The interface is defined in C, which is the programming
language supported by the MVP, and a short summary of the available functions is given in Table 8.1.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suspend, Yield,</td>
<td></td>
</tr>
<tr>
<td>GetTaskStatus, WakeTask</td>
<td>Synchronization</td>
</tr>
<tr>
<td>GSHMMalloc, FindGSHM, GSHMFree</td>
<td>Shared memory support</td>
</tr>
<tr>
<td>GetTime, SleepMS, MakePeriodicTask,</td>
<td></td>
</tr>
<tr>
<td>WaitNextPeriod</td>
<td>Scheduling</td>
</tr>
<tr>
<td>DisplayRenderPixel, DisplayRenderText</td>
<td>Virtual peripheral support</td>
</tr>
</tbody>
</table>

Table 8.1: MVP API Functions

It can be seen that advanced features like message queue are not available in the MVP, and the functions given are primitive in terms of their functionality. This is intended by the MVP for several reasons. First, the number of functions is kept small so that the programmer does not need much effort to learn them, and thus can easily start programming. Second, the functionality is generic and simple, through which the application source code can be kept as much MVP independent as possible. Once the target platform is available, the code developed on the MVP should be able to be reused with a minimum amount of additional effort.

The virtual peripheral functions are provided to visualize data in graphic and text format. For programmers, this is not the only way supported by the MVP to get output from the application running in the simulator. Host I/O like `fread` and `printf`, can be directly used by native tasks without changing the code.

### 8.4.2 Code Generation

Given the MVP API, writing software for running in the MVP simulator is mostly like normal C application development, except that the code generation flow is different from that of a host application. Instead of being compiled to native executables, tasks must be given to the simulator as shared libraries. Since the simulator itself does not contain software behaviors, they are brought into the simulation through these libraries.

One important information which is determined during the code generation process is the timing of the application. Due to the fact that a piece of C code itself does not give any hint about how much time it needs to execute, it is necessary to use external tools to determine the timing information. Currently, the MVP tool chain supports three different flows to do code generation for MPSoC applications, namely, native, virtual cross-compilation and instruction-set simulation. Figure 8.7 gives an overview of all three approaches.

- **Native**: the source code is instrumented before being compiled into native binaries. Since the instrumentation does not consider the processing elements in the potential MPSoC platform, the flow is completely target independent and fast, but the timing information it introduces is the least accurate.

- **Virtual Cross-Compilation**: the source code is first processed by a special virtual cross-compiler which is similar to a cross-compiler but processes the code natively, then compiled into native binaries. With this flow, the performance estimation of the application is improved in comparison to the native flow, because the virtual cross-compiler takes the target
processor information into account. Nevertheless, the execution speed of the application is lower.

- **Instruction-Set Simulation**: in this flow, a cross-compiler is used to translate the source code into the target binary. It is then loaded into an ISS which is wrapped in a dynamic library. The previous two flows are not compatible to the target MPSoC platform, this approach allows the programmer to develop code which is binary compatible to the target. However, since applications are executed in an ISS whose speed is normally much slower than the native execution speed, this flow is the slowest one in the three approaches.

### Native Flow

The native flow supported by the MVP provides developers a processor independent way to run their applications. In order to allow user applications to run in the SystemC based simulator, several issues need to be solved here. First, C applications have their own `main` functions, which will conflict not only with the one in the SystemC kernel but also with each other if they are directly linked together. This problem is solved in the MVP by compiling the applications into shared libraries and using the task modules in the simulator as loader to load them dynamically before the simulation starts. During the dynamic load process, the functions in the libraries are all recognized as extern, hence no conflict will occur even if there is a function having the same name as one in the simulator.

Moreover, it is mentioned earlier that pure C code is natively executed in a SystemC simulation without advancing the clock, i.e. no time is spent in the code, which does not reflect the real behavior of the application. Normally, programmers are required to manually annotate the source code by inserting calls of the SystemC `wait` function in order to let the software consume simulation time. However, since such approaches introduce extra constructs, which are irrelevant to the application, into the source, the reusability of the code is reduced. For this reason, the MVP provides a user transparent and target independent solution for the code generation of native tasks. Figure 8.8 gives an overview of the tool flow.

The tools are developed based on the LLVM compiler framework [141]. The application source code is first parsed by the LLVM C frontend and optimized if required by the programmer. Afterwards, an MVP code instrumentor is used, which automatically inserts calls to the `step` function between basic blocks for consuming simulation time. The behavior of the `step` function is very simple. It first consumes time and then checks if the time slot granted by the VPE is used up. In case of yes, it will switch the status of the task to `READY`, send out an event to inform the VPE of the...
change and start waiting for the next TASK_RUN event. Otherwise, the execution of the task just continues.

For the computation of the elapsed simulation time, since the native flow is focused on high-level functional simulation and no specific processor information is available here, the calculation is simplified by assuming that each VPE uses one clock cycle to perform a C level operation like an addition. The instrumentation is done per basic block so that a very high simulation speed can be achieved. It needs to be mentioned that using a cost table as in the profiling process is also possible in the native flow. Nevertheless, since the Virtual Cross-Compilation flow can already better estimate the application performance than the cost table approach [142], it is not employed in the current MVP.

Finally, it needs to be mentioned here that the whole flow is automatic and does not require any manual code annotation. The C source code is kept intact and thus can be potentially further reused for the target platform. In the simplest case, a sequential application can be brought to the MVP through just replacing the native compiler with the MVP tool chain.

**Virtual Cross-Compilation Flow**

The virtual cross-compilation flow uses a special code transformation module called *virtual compiler backend* which is initially introduced in the Totalprof framework [142]. The MVP uses the backend to improve the accuracy of software timing. An overview of the tool flow is given in Figure 8.9.

The first two steps in the tool flow do the same thing as their counterparts in the native flow. The input C source code is translated by the frontend into an Intermediate Representation (IR), and machine independent optimizations are performed on the IR. Afterwards, the optimized IR is processed by the virtual compiler backend which is a special IR-to-IR transformation module. It has all components necessary for a cross compilation process such as code selection, register allocation, scheduling, etc. Nevertheless, the code produced by the virtual compiler backend is not machine assembly but LLVM IR, which is called virtual assembly here. Since the virtual assembly is very similar to the target assembly, performance estimation can easily assess the timing information of each basic block from it. Besides, the potential micro-architecture level events such as pipeline stalls are taken into account during the analysis. The resulting performance information is then used by the following MVP instrumentation module to insert the *step* function calls which later consume simulation time in the simulation. Finally, a native compiler backend is used to
generate native shared libraries.

Since the virtual compiler backend only performs IR-to-IR translation and the final outputs of the whole flow are still native, a high simulation speed can be achieved (usually more than 1,000MIPS for a single processing element [142]). Considering that both the instruction set and the micro-architecture of the target processor are taken into account by the performance estimation, the high speed does not come at the cost of accuracy.

The virtual compiler backend is target processor dependent. For each specific processor architecture, a corresponding backend needs to be developed. The backend is template-based, and all its components are retargetable. The retargeting process is driven by a processing element description, which supports different types of processor architectures, such as RISC, DSP, VLIW, and ASIP. Overall, the required effort is significantly less than traditional compiler retargeting, because the generated code is still native and no rigorous Application Binary Interface (ABI) level compatibility for the target processor has to be met.

**Instruction-Set Simulation Flow**

The third alternative provided by the MVP to run an application is using an ISS. This is supported mainly for the early MPSoC design stage when the overall architecture of the target platform is still under design, but the use of certain processor is already determined. Besides, the combination of the ISS tasks and the abstract OS model in the VPE allows the developer to early evaluate the application behavior as if an OS is available, even though the real OS for the target processor would be finished much later in the design. Of course, for both cases, an ISS is needed beforehand.

The code generation procedure of the ISS tasks is the same as a typical target-compilation flow, where a cross-compiler is used. Since the target binary will be executed by an instruction-set simulator, assembly code is also allowed here, which gives the programmer more flexibility in writing the application.

Nevertheless, in order to control the instantiation and the execution of the ISS, an extra wrapper is required, which is mainly responsible for instantiating the ISS, stepping the ISS according to the event sent from the VPE and rerouting all accesses to the shared memories to the corresponding GSHM blocks.
8.5 Debug Method

One advantage of using the MVP for parallel application development is that debugging is much simpler than that in a real multiprocessor machine. Since the simulation is based on SystemC and executes in one thread, the behavior of the application is independent from the host machine and thus is fully deterministic. Besides, there is no need for special debuggers in order to debug natively executed tasks. The instrumentation process in their code generation flow does not destroy the debug information. To debuggers, they look just like normal host applications. Therefore, the programmer can use any host debugger like GDB to connect to the simulation thread and debug the code.

For ISS tasks, the problem is a little bit more complicated. Their debugger support depends on the used ISS and its wrapper. The ISS must first support the connection from a debugger; and the wrapper needs to be implemented in a way that it can block the simulation and wait for the connected debugger to finish the user interaction. The LISATek generated simulator and the corresponding wrapper fulfill the above mentioned requirements. Thus, tasks running in the ISS can also be debugged in the MVP. In a mixed simulation, where ISS tasks are executed together with native tasks, both the ISS debugger and the host debugger can be used concurrently as depicted in Figure 8.11. The host debugger is directly connected to the simulator for debugging native tasks; at the same time, the debugger for the ISS is connected through a channel which is instantiated by the wrapper.

In addition to the source-level debugging, the MVP also provides facilities for monitoring system
level events occurred in the simulator. For example, the user can use the GUI to enable the generation of VCD trace files which record the time and the duration of the activation of all tasks. This gives the developer an overview of the application execution process.

8.6 Summary

High-level simulation is employed in the MAPS framework mainly for early MPSoC software development. It is supported by the MAPS virtual platform, which consists of a generic MPSoC simulator and a software tool chain. The former models MPSoC architectures at a very high abstraction level, and the latter translates the application source code into binary executables which are dynamically loaded into the simulator. For MPSoC programmers, the MAPS virtual platform provides the possibility of simulating applications early to verify their functionality. Besides, a rough performance estimation can also be obtained. In the next chapter, the use of the high-level simulation will be presented through case studies.
This chapter presents some results which demonstrate the applicability and efficiency of the MAPS methodology proposed in this thesis. Two applications in the multimedia domain and several MPSoC platforms are used as target. The case studies are carried out with the following objectives in mind:

- To demonstrate how the tools provided by the MAPS framework can help programmers with the development of MPSoC applications, and
- To show the efficiency of the framework by examining the performance of the resulting applications on target platforms.

The rest of this chapter is organized as follows: the first case study uses a JPEG [143] encoder as target application, Section 9.1 shows how it is parallelized in the MAPS framework for the TCT [73] MPSoC. Section 9.2 shows the result of a parallel H.264 baseline decoder for several different MPSoC platforms. Finally, Section 9.3 summarizes this chapter.

### 9.1 JPEG Encoder

JPEG (Joint Photographic Experts Group) is a widely used compression method for photographic images. The degree of compression can be adjusted, allowing a selectable trade-off between storage size and image quality. In this case study, a JPEG encoder application is used, whose sequential implementation in C is available at the beginning as starting point. It is a modified version of the encoder in a free JPEG library developed by the Independent JPEG Group [144]. As mentioned in Section 2.3.10, the TCT MPSoC platform [73] is used here as target.

#### 9.1.1 TCT Tools

As a backend in the MAPS development flow, the TCT platform is responsible for binary code generation, instruction-set simulation and parallel performance profiling [73]. This case study
mainly uses the following TCT tools:

- **TCT Compiler** which takes C codes with \texttt{THREAD} annotation as input, analyzes interprocedural dependence flows including pointer dereferences, inserts thread communication instructions for thread activation, data transfer and data synchronization, and finally generates partitioned thread executable binaries. Its concurrent execution model is constructed upon a hierarchy of functional pipelining and task parallelism for a fully distributed memory system which guarantees race-free, deterministic behavior.

- **TCT Simulator** which consists of a set of cycle-accurate instruction-set simulators for parallel thread execution. The communication on the full crossbar interconnect is also modeled at cycle accuracy. For convenience, the number of processors can be configured in the simulator and the user can instantiate more than 6 processors for testing purposes.

- **TCT Trace Scheduler** which is a fast performance estimation tool. It uses an abstract computation/communication model to quickly report the estimated execution time with accuracy of few % error from the TCT Simulator.

The TCT platform used in the case study does not support sharing one processor with multiple parallel threads. Due to this limitation, the temporal and spatial task mapping can be easily solved, i.e. one task per processor.

### 9.1.2 Profiling

To develop a parallel version of the encoder, the sequential one is first profiled in the MAPS framework. For this purpose, the processing element cost table of the MAPS architecture description is adapted to the TCT architecture in order to reflect the corresponding computation cost.

Figure 9.1a shows an overview of the call graph of the sequential JPEG encoder. Since the node color in the graph is related to the share of each function in the overall execution cost, one can directly identify those heavily weighted functions. In Figure 9.1c, the 5 most time consuming functions are listed along with their numbers of calls and execution costs. Among these functions, \texttt{JPEGtop} is the most interesting one, because it is invoked only once by the \texttt{main} function but occupies almost 100\% of the overall execution cost. Besides, it directly calls the \texttt{BLK8x8} and \texttt{ReadOneLine} functions which are also in the list. Figure 9.1b gives an enlarged view of part of the call graph, which shows the relation between \texttt{JPEGtop}, \texttt{BLK8x8} and \texttt{DCTcore}. Based on the profiling information provided by MAPS, it can be seen that the focus of parallelization should be put on the \texttt{JPEGtop} function. Nevertheless, to explore the most parallelism out of the application, other computation intensive functions in the list are also passed to the MAPS partitioning tool.

### 9.1.3 Partitioning

To obtain an initial partitioning of the application, the MAPS automatic partitioning tool is used. For the \texttt{JPEGtop} function, it generates 5 tasks which cover all the hot spots in the function. Figure 9.2 is a screen shot of the MAPS IDE showing the partitioning result. It can be seen that all the hot spots of the function are recognized automatically by the tool, and tasks are generated which cover the corresponding code blocks.
In the other top 5 functions, the DCTcore function is not partitioned, because the Discrete Cosine Transformation (DCT) it performs is known to be done mostly as an atomic transformation on data. Besides, the main function is also excluded from partitioning, for it simply calls the JPEGtop function and does not perform further control or computation. Table 9.1 gives a summary of the tasks in the initial partition created by the MAPS partitioning tool.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEGtop</td>
<td>5</td>
</tr>
<tr>
<td>BLK8x8</td>
<td>2</td>
</tr>
<tr>
<td>ReadOneLine</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 9.1: Initial Partition of the JPEG Encoder Application

Generating target code from tasks for the TCT platform is very straightforward. A code generator is developed, which automatically inserts the TCT THREAD annotation around those code blocks which belong to tasks. The MAPS user can directly invoke the TCT code generator through the MAPS IDE.
9.1.4 Simulation Result and Further Refinement

To examine the performance of the parallelized encoder, the TCT simulator is used to execute the code. Before that the code generated by MAPS needs to be first compiled by the TCT compiler. The initial partition produced by the MAPS tools gives 3.61x speedup using 16 processors, which achieves 0.23 parallel efficiency. Here, parallel efficiency, $E(P)$, for an application running on $P$ processors is defined in the usual way \[145\] by

$$E(P) = \frac{\text{speedup}}{P}$$  

By examining the graphical execution traces provided by the TCT trace scheduler, one can find out that another THREAD annotation is needed to obtain a hierarchical pipeline (step 2). Since this is a special feature of TCT which is not commonly available in other MPSoCs, the MAPS partitioning tool currently does not automatically exploit this potential. However, the programmer can manually modify the automatic partitioning result through the MAPS IDE in order to improve it. After the second step, the speedup of the application is raised to 5.48x by using 17 processors, i.e. 0.32 parallel efficiency. In comparison, a manually partitioned JPEG encoder version is included in the TCT tools which achieves 9.43x speedup using 19 processors. Table 9.2 summarizes the speedup results.

<table>
<thead>
<tr>
<th>Partition</th>
<th>Speedup (x)</th>
<th>Number of Processors</th>
<th>Parallel Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>3.61</td>
<td>16</td>
<td>0.23</td>
</tr>
<tr>
<td>step 2</td>
<td>5.48</td>
<td>17</td>
<td>0.32</td>
</tr>
<tr>
<td>Manual</td>
<td>9.43</td>
<td>19</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 9.2: Speedup of the JPEG Encoder Application

Finally, it needs to be mentioned that the whole parallelization using the MAPS framework is carried out in around two hours, and Figure 9.3 shows the time spent by the MAPS tools in a PC.
workstation with a 2.67GHz quad-core CPU and 8GB RAM. It can be seen that the execution of the MAPS tools only takes a small fraction of the overall parallelization time. Besides, the 10 THREAD annotations out of the total 12 annotations generated by the MAPS partitioning tool are also recognized as tasks in the manual partition. Only the BLK8x8 function is partitioned by the MAPS user differently than the manual partitioning does, which is the cause for the speedup gap between the manual and the semi-automatic partition. This is mainly due to the complex data dependencies between different executions of this function across different loop iterations. A possible solution to overcome this problem is to extend the data dependency analysis, e.g. by taking the loop information into account.

A possible solution to overcome this problem is to extend the data dependency analysis, e.g. by taking the loop information into account.

![Diagram]

Figure 9.3: Time Spent by the MAPS Tools

9.2 H.264 Baseline Decoder

The second case study is an H.264 [146] baseline decoder. H.264 is a standard for video compression, which is able to achieve a high compression ratio through the use of algorithms with high computational complexity. The goal of this case study is to develop a parallel H.264 baseline decoder for two MPSoC platforms, Multi-ARM platform and Multi-LTRISC/LTVLIW platform. Like the previous case study, a sequential C implementation of the decoder is also given as the starting point of the case study.

9.2.1 Target Platform

Two MPSoC platforms are used in this case study as target:

- **Multi-ARM**, which is a homogeneous MPSoC consisting of multiple ARM926-EJS [81] processors;
- **Multi-LTRISC/LTVLIW**, which employs a combination of RISC and VLIW processors.
Multi-ARM

The Multi-ARM platform is an MPSoC platform which is developed in-house [80]. It has a configurable number of ARM926-EJS processor cores and some peripheral modules for tasks like I/O, display, etc. A big chunk of shared memory is used for both instruction and data of all the ARM cores in the MPSoC. The interconnection between the components are realized through an AMBA bus [147]. Since the platform is still experimental, no hardware prototype is built for it. In order to test the application and measure the parallel performance, a SystemC based virtual prototype is built by using the CoWare Virtual Platform [52] tool suite, which simulates the functionality of the experiment platform. The ARM processor is modeled with instruction accurate ISS in the virtual prototype. An overview of the platform is shown in Figure 9.4.

![Figure 9.4: Multi-ARM Platform](image)

Within the platform, a light-weight OS is responsible for scheduling the parallel tasks dynamically. On top of that, tasks access the low level OS services through API functions. The communication between tasks is realized by using the shared memory. The operating system provides semaphore support for synchronizing shared memory accesses.

Unlike the TCT platform which has a compiler capable of compiling C code with THREAD annotations, the Multi-ARM platform does not have such advanced compiler support. Therefore, for this platform, the transformation from tasks identified by the MAPS framework to target C code with the target OS API is done manually.

Multi-LTRISC/LTVLIW

The Multi-LTRISC/LTVLIW platform is another in-house experimental MPSoC platform. It uses two processing elements as basic building blocks, the LTRISC and LTVLIW processor which are both taken from the CoWare IP library. The former is a simple RISC processor with 5 pipeline stages, and the latter is a 4-issue VLIW processor using the same basic instruction set as the former. The platform is configurable, and the number of the RISC and VLIW processing elements can be adjusted at design time according to requirement. Each processor in the platform has a local memory for storing instructions and data, and the platform has a block of memory shared between all processors. All components of the platform are connected by using a SimpleBus which is a simple bus model coming with the OSCi SystemC kernel [119]. Two variants of the platform are used in this case study, Multi-LTRISC and Multi-LTRV. The former is a homogeneous MPSoC with all its processing elements being the LTRISC processors, Figure 9.5a illustrates its basic
structure. The latter is configured to be a heterogeneous MPSoC with one LTRISC processor and the rest being the LTVLIW processor as Figure 9.5b shows. Like the Multi-ARM platform, virtual prototypes are built for testing and benchmarking the target application, in which processors are simulated at cycle accuracy.

![Diagram of Multi-LTRISC/LTVLIW Platform](attachment:image.png)

**Figure 9.5: Multi-LTRISC/LTVLIW Platform**

The Multi-LTRISC/LTVLIW platform does not have operating system support. Therefore, no temporal mapping is possible in this platform, the task-to-processor mapping is simplified to one task per processor. Tasks communicate with each other through the shared memory. Since no automatic target code generator is available, the code generation for the platform is done manually.

### 9.2.2 Profiling

The process of profiling is similar to that of the previous case study, except that the cost table in the MAPS architecture description is adapted to the Multi-ARM and Multi-LTRISC/LTVLIW platform respectively. The LTVLIW processor is not taken into account, because the profiling method currently implemented in MAPS does not support processors with ILP well, and this will be improved in future developments by using tools like the Totalprof [142]. In the case study, the LTRISC processor is used for profiling the application for the Multi-LTRISC/LTVLIW platform.

Figure 9.6a lists the 5 most time consuming functions of the application, and Figure 9.6b shows the calling relationship between them. Note that, the profiling result of the Multi-ARM platform is very close to that of the Multi-LTRISC/LTVLIW platform. In fact, the difference of the function total cost in percentage is smaller than 1%, which is negligible in the selection of hot spots. The results of two platforms are therefore shown in Figure 9.6a together in one column. This can be explained by the fact that the LTRISC processor and the ARM9-EJS processor both feature a RISC instruction-set and have a similar pipeline architecture.

Based on the obtained profiling information, it can be seen that the `DECODEH264Start` function controls the execution of decoding. Both the syntax decoding and Macro-Block\(^1\) (MB) decod-

\(^1\)In the H.264 standard, a Macro-Block is defined as a 16x16 pixel block in a video frame.
ing are invoked by the function. Therefore, the parallelization of the decoder is focused on the `DECODEH264Start` function.

### 9.2.3 Partitioning

The initial partitioning produced by MAPS contains three parallel tasks, the `main` task, the `Syntax Decoder` task, and the `MB Decoder` task, among which the latter two are located in the `DECODEH264Start` function and activated in order by the first one. Figure 9.7a) shows the relation of the three tasks. Due to the sequential execution of the `Syntax Decoder` and the `MB Decoder`, the parallelism that can be explored by this partitioning, is limited.

To further exploit the parallelism, the source code is manually analyzed, through which it is found that the body of the `MB Decoder` task is a loop which invokes the `DecodeMacroBlock` function once each iteration to decode a macro block. Since macro blocks in a video frame can be decoded in parallel, the loop is unrolled to enable parallel decoding of MBs. Given the information from the H.264 standard and a QCIF (widthxheight=176x144) sized input video sequence, it can be known that decoding 6 MBs simultaneously is possible. Therefore, 6 `MB Decoder` s are instantiated in the improved partitioning. Besides, the `main` task and the `Syntax Decoder` are merged, because they execute one after the other, and it is not necessary to separate sequential executed code. Figure 9.7b) illustrates the partitioning result after the improvement, which is in fact an instance of the master/slave parallel programming paradigm. The `Main` task is the master, which controls the execution of a number of slave `MB decoders`.

### 9.2.4 Target Code Generation

Unlike the TCT MPSoC, the target platforms used in this case study do not have automatic code generation support which allows translating the MAPS generated tasks to target code automatically. Therefore, the target code generation in this case study is done manually. The behavior of each task is first extracted from the sequential application and encapsulated as the target platform requires. The Multi-ARM platform requires that a task must be wrapped in an entry function which is later called by its OS, and each tasks in the Multi-LTRISC/LTVLIW platform is compiled as a standard alone application. Besides, the number of MB decoders is implemented to be
configurable, because the target platforms are both configurable and can use different numbers of processors.

About one man week is spent in writing target code for the MPSoC platforms used in this case study. In comparison, the MAPS tools just need minutes to produce the profiling and the partitioning results for the application. The reason for the time consuming manual code writing process is mainly the data dependence analysis and the corresponding code modification. Especially, when the declaration, initialization, communication and manipulation of data objects are distributed over different functions and source files, the analysis is very time consuming.

An example of such scenario written in pseudo C code is shown in Figure 9.8. Suppose function_3

```c
void function_3(data_struct* struct_pointer);
...
void function_2(...){
    //function_3 is supposed to be a parallel task run in another processor
    function_3(struct_pointer);
    ...
}
```

File_1.h - Declaration

```c
data_struct* new_struct();
...
function_1 {
    ...
    data_struct* struct_pointer = new_struct();
    ...
    struct_pointer->buffer_1 = malloc(...);
    ...
}
```

File_2.c - Initialization

```c
void function_3(data_struct* struct_pointer);  ...
void function_2(...){
    //function_3 is supposed to be a parallel task run in another processor
    function_3(struct_pointer);
    ...
}
```

File_3.c - Communication

```c
//Code inside the parallel task
...
function_x(...)
{
    ...
    read_buffer(struct_pointer->buffer_1);
    }
    ...
```

File_4.c - Manipulation

Figure 9.7: Partitioning Result of the Sequential H.264 Baseline Decoder

Figure 9.8: Manual Data Dependency Analysis
in File 3.c is going to run in a separate processor, which implies that its input parameter, `struct_pointer`, needs to be communicated between processors. Since the pointer points to a data structure which is declared in File 1.h and initialized in File 2.c, simply sending the pointer value is not enough. The complete structure needs to be copied from one processor to another. Unfortunately, sometimes, this is still not enough, because the data structure can contain more pointers addressing dynamically allocated data, e.g. `data_struct` itself has two more pointers, which store the addresses of two buffers. In File 4.c, one buffer is accessed by `function_x` in the parallel task. To allow parallel execution, such data dependencies need to be found completely, and extra code is needed to transfer data between processors. The example is kept simple for the sake of clarity. In reality, a data structure can be accessed at many places in thousands of lines of code, and it is not trivial to discover all the data dependencies.

In this case study, the dependencies are still resolved manually for writing the target code. Nevertheless, it is desirable to have a tool which can help the developer with the work. The MAPS framework might provide such a tool in future.

### 9.2.5 Simulation Results

The simulation of the parallel decoder is done in two steps. First, the MAPS virtual platform (MVP) is used to run the parallelized decoder at high-level with native simulation techniques, because the simulation speed of VPs is limited by ISSes use and it is easier to debug code in a host machine than in a VP. Besides, the performance of the parallelized decoder can also be roughly estimated with the MVP. Afterwards, target VPs are employed to finally check if the parallelized decoder is correctly implemented and the performance predicted at high-level also holds true in targets.

#### High-Level Simulation

The functionality of the parallelized decoder is first tested by decoding an encoded H.264 video file and checking the output video frame in the virtual display of the MVP. Once the functional test is passed, the speedup achieved by the parallelized decoder is measured in order to obtain an early evaluation of the effect of parallelization.

The parallelized decoder is simulated in the MVP using two techniques, native execution and virtual cross-compilation (VCC). Note that, the native execution mode of the MVP does not directly support VLIW processors. In the case study, the clock frequency of the LTVLIW processor in native mode is adjusted to 1.7 times as high as the LTRISC processor so as to mimic the speedup brought by the Instruction Level Parallelism (ILP). The number is given by the developer of the LTVLIW compiler, who knows that the processor is able to execute 1.7 instructions per cycle (IPC) on average, which is about 1 in case of the LTRISC processor. In the VCC mode, this frequency scaling is not necessary, because the virtual compiler backend already takes the ILP effect into account. Figure 9.9 shows the application speedup measured by using the MVP, where the speedup value is calculated by the following equation:

\[
\text{Speedup} = \frac{\text{Time}_{\text{Sequential}}}{\text{Time}_{\text{Parallel}}} \tag{9.2}
\]

The curves labeled with `Native` are obtained through native execution, and those with `VCC` in
Figure 9.9: Application Speedup Measured in the MAPS Virtual Platform

their labels are simulated using the virtual cross-compilation technique. The SMP Native curve estimates the speedup of the parallelization when identical processing elements are used in the target platform. This corresponds to the Multi-ARM platform and the Multi-LTRISC platform. The Multi-LTRISC VCC curve shows the result by using the LTRISC virtual compiler backend, which is a bit biased from that of the native execution due to the different performance estimation method. The Multi-LTRV Native curve shows the native estimation result which is very close to the result from using the virtual cross-compilation (Multi-LTRV VCC).

The results show that, in general, the speedup increases when more processors are used. Nevertheless, it is not necessary to use more than 4 processors, when they are the same. Two potential causes could be responsible for the saturation of the speedup, not enough parallelism or not fully explored parallelism. The former means that the application itself does not have more potential for parallel execution; the latter means that the application can be further parallelized. Since the major goal of the MVP is functional test, further exploration for improvement is done latter with virtual platforms, which can do performance estimation more accurately.

Virtual Platform Simulation

The MVP provides an early development environment in which parallel application can be simulated and tested. Finally, the application needs to be executed on target platforms. The results are shown in Figure 9.10. As reference, a fully manually parallelized version for the Multi-ARM platform is available, and the curve Multi-ARM Manual shows its speedup result. The other curves are results from the MAPS parallelized versions.

Compare to the curves in Figure 9.9, it can be seen that, in general, the results are quite close, which implies that the MVP has shown the correct trend of the application speedup. Nevertheless, the MAPS parallelized decoder shows a smaller speedup in the Multi-ARM virtual platform. This can be explained by the fact that the OS overhead of the Multi-ARM platform is not taken into account in the MVP.

Besides, in comparison to the manually parallelized version, the MAPS parallelized decoders show
smaller speedup on the Multi-LTRISC platform and the Multi-ARM platform. A better speedup is achieved only with the help of the instruction level parallelism of the LTVLIW processor, which implies that there is still potential for further performance improvement. To find the reason for the better performance, a close study of the manual version is made, which tells that a dynamic MB task dispatcher is the major source of the difference. According to the H.264 standard, there can be dependencies between the MBs in H.264 video frames. In the MAPS parallelized decoders, a conservative assumption is made that the MBs always depend on their neighbors, which is not completely true but simplifies the analysis. In the manual version, the MB dependency is dynamically resolved through analyzing the information encoded in the MB header, and an MB task is dispatched as soon as its depending MBs are decoded. This reduces the time spent by processing elements in waiting, and therefore results in better performance. However, exploring such opportunities requires deep application knowledge and time. It is estimated that one man month is needed to parallelize the decoder for the Multi-ARM platform. Consider that the MAPS parallelized decoders for the three platforms are developed in about one man week, it can be said that the development efficiency is improved and the results are acceptable.

9.3 Summary

In this chapter, the results of two case studies are introduced in detail, in which the MAPS framework is used throughout the entire software development flow for several MPSoC platforms. For the TCT platform, the JPEG encoder is efficiently parallelized using the MAPS tools. Besides, parallel H.264 decoders are developed in the framework for multiple targets. In both cases, the proposed method has improved the development efficiency.
Chapter 10
Summary and Outlook

10.1 Summary

Software development is one of the most challenging tasks in MPSoC design. Therefore, tool support is demanded by developers in order to improve the efficiency of MPSoC software development. For this purpose, the MAPS framework is proposed and developed within the range of this thesis.

MAPS provides MPSoC software developers a unified environment integrating all necessary tools that are needed for MPSoC development. The development process is divided by MAPS into a sequence of steps: application modeling, architecture modeling, profiling, control and data flow analysis, partitioning, high-level simulation and code generation. The tool chain can be connected together to form a highly automated development flow. Meanwhile, MAPS also allows the developer to interact with the environment between each step, so that he can always fine tune the intermediate result with his knowledge. In contrast, an automatic parallelizing compiler is a monolithic software tool with less user interaction. The philosophy behind the MAPS framework is providing assistance to developers and involving them in the development flow, which can be seen from the following aspects:

- At the early stage of the development, the MAPS profiling facility helps MPSoC software developers to better understand the target application, so that potential parallelism can be quickly identified. The visualization of the profile information is seamlessly integrated into the MAPS environment, and the developer can see the result directly in an intuitive way.

- For partitioning the target application, MAPS provides suggestions to MPSoC software developers on where parallel tasks can be created. To achieve this, the control and data dependence inside the application is thoroughly analyzed by using both traditional compiler techniques and dynamic analysis. The latter is implemented in MAPS to compensate the former such that precise dependence information about the application can be obtained. It is based on this information, that MAPS searches for parallel tasks for the programmer on a novel granularity level. Parallel tasks are identified by MAPS according to their internal and external control and data dependency. When necessary, the programmer can also have
the complete control over the partitioning process through interacting with the tool.

- Once a partitioning is available, the MAPS virtual platform (MVP) helps the developer to test the result early at a high-level abstraction level. Native execution is used, which ensures a fast simulation speed. With the MVP, MPSoC software developers can easily explore different partitioning and application scenarios.

- To run the parallelized application on the target MPSoC platform, target code needs to be generated. The MAPS framework aims to completely take over this process and generate target code fully automatically. Within this thesis, a code generator for the TCT MPSoC is developed. It produces code that can be directly compiled by the TCT compiler.

The usability of the MAPS framework is shown by two case studies, a JPEG encoder and an H.264 baseline decoder. With the help of the MAPS tools, the former is successfully parallelized for the TCT MPSoC. Although the achieved speedup is still behind what can be achieved manually by a programmer experienced with the TCT compiler and architecture, MAPS helps the programmer to obtain a reasonably parallelized version within a short period of time. Since the framework is fully under the control of the developer, further improvement is possible when the user has more knowledge and experience with the platform.

In the second case study, several MPSoC platforms are used as target, for which parallel H.264 decoders are developed. By the MAPS tools, parallel tasks are identified within the sequential version of the target application. Based on the MAPS suggested partitioning, the application is parallelized. Before the parallel decoder is tested on target platforms, the MAPS virtual platform (MVP) is employed as an early simulation vehicle to perform functional tests and estimate the effect of parallelization. Finally, the parallelized application is executed on the target MPSoC platforms, and the simulation results show that the effect of parallelization is well predicted by the MVP.

The results from the case studies show that the efficiency of MPSoC software development is improved by MAPS through its advanced tooling support and high level of integration. With MAPS, programmers are able to develop code for MPSoCs quickly.

10.2 Outlook

MPSoC software development is an area being actively researched. Within this thesis, the concept of the MAPS framework is developed and a set of tools are developed to realize the proposed methodology. Compared to other academic works, which are mostly composed of sparse connected tools, MAPS targets providing developers a single complete environment for application modeling, source analysis, code partitioning, scheduling/mapping, target simulation, and debugging. To achieve this goal, future developments are definitively required.

This thesis uses sequential C as input specification in which the parallelism of the target application is implicit. Sophisticated analyses are required to extract such implicit parallelism from sequential code. To improve this, application models with explicit parallelism such as the Kahn Process Network can be used. This is a feature that will be included in the future MAPS development.

Due to the limitation of the target MPSoC, scheduling and mapping is solved in a relatively simple way in this thesis. In the future, for complex MPSoC platforms with advanced parallel runtime supports, further exploration is needed to find a viable solution for both problems.
Moreover, target code development is always a tedious work for MPSoC programmers. In this thesis, some code generation work is still done manually. To have a more fluent MPSoC software development flow, a lot of work needs to be done to further automate the code generation process for different MPSoC platforms.

Last but not least, debugging is another important step in software development after the executable code is generated. Especially for MPSoCs, which run parallel applications, debugging is a more challenging work than for uniprocessor systems. New debugging methods and tools are especially appreciated by MPSoC programmers when they can help to solve the puzzle of MPSoC software bugs.
Appendix A

MAPS Architecture Description XML Schema

A.1 Component Library

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:archlib="ARCHLIB"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="ARCHLIB">
    <xsd:element name="ArchitectureLib" type="archlib:ArchitectureLib"/>
    <xsd:complexType name="ArchitectureLib">
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="1" name="ProcessingElement" type="archlib:ProcessingElement"/>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="CommunicationChannel" type="archlib:CommunicationChannel"/>
        </xsd:sequence>
        <xsd:attribute name="Name" type="xsd:string"/>
    </xsd:complexType>
    <xsd:complexType name="ProcessingElement">
        <xsd:sequence>
            <xsd:element name="CoreType" type="xsd:string"/>
            <xsd:element name="Category">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="RISC"/>
                        <xsd:enumeration value="CISC"/>
                        <xsd:enumeration value="VLIW"/>
                        <xsd:enumeration value="SIMD"/>
                        <xsd:enumeration value="GPP"/>
                        <xsd:enumeration value="DSP"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```
<xsd:complexType name="CostTable">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1" name="Operation">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element maxOccurs="unbounded" minOccurs="1" name="VariableType">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="Cost" type="xsd:float"/>
                            </xsd:sequence>
                            <xsd:attribute name="Name" type="archlib:Vartype" use="required"/>
                        </xsd:complexType>
                    </xsd:element>
                    <xsd:attribute name="Name" type="archlib:Optype" use="required"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
<xsd:enumeration value="LessEqualThan"/>
<xsd:enumeration value="GreaterEqualThan"/>
<xsd:enumeration value="Equal"/>
<xsd:enumeration value="NotEqual"/>
<xsd:enumeration value="And"/>
<xsd:enumeration value="Xor"/>
<xsd:enumeration value="Or"/>
<xsd:enumeration value="Cast"/>
<xsd:enumeration value="Constant"/>
<xsd:enumeration value="Call"/>
<xsd:enumeration value="Assignment"/>
<xsd:enumeration value="ConditionalJump"/>
<xsd:enumeration value="Jump"/>
<xsd:enumeration value="ReturnValue"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="Vartype">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Char"/>
    <xsd:enumeration value="Short"/>
    <xsd:enumeration value="Int"/>
    <xsd:enumeration value="Long"/>
    <xsd:enumeration value="LongLong"/>
    <xsd:enumeration value="Float"/>
    <xsd:enumeration value="Double"/>
    <xsd:enumeration value="LongDouble"/>
    <xsd:enumeration value="Pointer"/>
    <xsd:enumeration value="Other"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="Protocol">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="P2P"/>
    <xsd:enumeration value="DMA"/>
    <xsd:enumeration value="BUS"/>
    <xsd:enumeration value="AHB"/>
    <xsd:enumeration value="DOL"/>
    <xsd:enumeration value="TCT"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="CostFunction">
  <xsd:sequence>
    <xsd:element name="Function" type="archlib:CFuncti0n" maxOccurs="1" minOccurs="0" />
    <xsd:element name="ChannelSetupTime" type="xsd:int" maxOccurs="1" minOccurs="0" />
    <xsd:element name="TransmissionSetupTime" type="xsd:int" maxOccurs="1" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
A.2 Architecture Model

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:arch="MAPSARCH" xmlns:archlib="ARCHLIB"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="MAPSARCH">
    <xsd:import namespace="ARCHLIB" schemaLocation="ArchitectureLib.xsd"/>
    <xsd:element name="Architecture" type="arch:Architecture"/>
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="1"
                name="Processor" type="arch:Processor"/>
            <xsd:element maxOccurs="unbounded" minOccurs="0"/>
</xsd:schema>
```
name="Channel" type="arch:Channel" />
<xsd:element maxOccurs="unbounded" minOccurs="1" name = "ProcessingElement" type = "archlib:ProcessingElement" />
<xsd:element maxOccurs="unbounded" minOccurs="0" name = "CommunicationChannel" type = "archlib:CommunicationChannel" />
</xsd:sequence>
<xsd:attribute name="Name" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="Processor">
  <xsd:sequence>
    <xsd:element name="Core" type="xsd:string" />
    <xsd:element name="Category">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="RISC" />
          <xsd:enumeration value="CISC" />
          <xsd:enumeration value="VLIW" />
          <xsd:enumeration value="SIMD" />
          <xsd:enumeration value="GPP" />
          <xsd:enumeration value="DSP" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Channel">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="1" name="Protocol">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Name" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:string" use="required" />
  <xsd:attribute ecore:reference="arch:Processor" name="Source" type="xsd:string" use="required" />
  <xsd:attribute ecore:reference="arch:Processor" name="Target" type="xsd:string" use="required" />
</xsd:complexType>

</xsd:schema>
Appendix B

Pseudo Code for Profiling

This chapter shows the pseudo code used by the profiling tools of the MAPS framework.

B.1 Code Instrumentation

\begin{verbatim}
For Each function in IR-C
  insertFunctionCall("EnterFunction")
  If function is main
    For Each globalVariable in IR-C
      insertFunctionCall("InitGlobalVariable")
    End For
  End if
  For Each localVariable in function
    insertFunctionCall("InitLocalVariable")
  End For
  For Each statement in function
    If statement is a basic block entry
      insertFunctionCall("EnterBasicBlock")
    End If
    If statement has memory access
      insertFunctionCall("TraceMemAccess")
    End If
    If statement calls function
      insertFunctionCall("PrepareFunctionCall")
      If statement call "malloc"
        insertFunctionCall("ProcessMalloc")
      End If
      If statement call "calloc"
        insertFunctionCall("ProcessCalloc")
      End If
      If statement call "free"
        insertFunctionCall("ProcessFree")
    End If
End For
\end{verbatim}
End If
End If
If statement is return
   insertFunctionCall("ExitFunction")
End If
End For
End For

B.2 Function Call Cost Calculation

Function FunctionCallCost (f, c, p)
Begin
   Result = FunctionSelfCost (f, c, p)
   For Each IR statement s in function F Do
      If s is a call statement which invokes function F’
         And F’ ≠ F Then
            Result = Result + FunctionCallCost (f’, s, p)
         End If
   End For
   Return Result
End

B.3 IR Statement Execution Cost Calculation

Function StmtExecutionCost (s, p)
Begin
   OverallExecutionCount = 0
   Result = 0
   b is the parent BB containing s
   f is the parent function containing b and s
   For Each IR statement c which calls f Do
      OverallExecutionCount += BBExecutionCount (b, f, c)
   End For
   Result += OverallExecutionCount*StmtIntrinsicCost (s, p)
   If s is a call statement which invokes function f’ Then
      Result = Result + FunctionCallCost (f’, s, p)
   End If
   Return Result
End
B.4 Call Graph Generation

Provide:

- $T$ is the input trace file;
- Function $GetNextTraceLine(T)$ returns one trace line a time in sequential order and returns $NULL$ at the end of $T$; and
- Function $GetEnteredFunction(t)$ returns the function an enter trace line enters.

Output:

- $CallGraph = (F, E, p)$

Procedure $GenDynamicCallGraph(T)$
Begin
  /* Construct the call graph structure, defined below*/
  ReadFunctionTrace($T$, NULL)
  /* Annotate node weights */
  For Each function $f$ in $F$ Do
    $W_{total}[f] = FunctionTotalCost(f, p)$
    $W_{self}[f] = 0$
    For each call site $c$ of function $f$ Do
      $W_{self}[f] = W_{self}[f] + FunctionSelfCost(f, c, p)$
    End For
  End For
End

Procedure $ReadFunctionTrace(T, current)$
Begin
  If current is not NULL Then
    If $F$ does not contain function current Then
      Insert current to $F$
      $W_{count}[current] = 1$
    Else
      $W_{count}[current] = W_{count}[current] + 1$
    End If
  End If
  $t = GetNextTraceLine(T)$
  While $t$ is not NULL And $t$ is not a function exit Do
    If $t$ is an entry trace line Then
      callee = $GetEnteredFunction(t)$
      If $E$ does not contain edge $e = (current, callee)$ Then
        Insert $e = (current, callee)$ to $E$
        $W_e[e] = 1$
      Else
        $W_e[e] = W_e[e] + 1$
      End If
ReadFunctionTrace(T, current, callee)

End If

t = GetNextTraceLine(T)

End While

End
Appendix C

MAPS Application Profile XML Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:profile="profile"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="profile">
    <xsd:element name="Profile" type="profile:Profile"/>

    <xsd:complexType name="Profile">
        <xsd:sequence>
            <xsd:element name="File" type="profile:File"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="File">
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" name="Line" type="profile:Line"/>
        </xsd:sequence>
        <xsd:attribute name="Name" type="xsd:string"/>
    </xsd:complexType>

    <xsd:complexType name="Line">
        <xsd:attribute name="Number" type="xsd:nonNegativeInteger"/>
        <xsd:attribute name="Cost" type="xsd:nonNegativeInteger"/>
        <xsd:attribute name="Percentage" type="xsd:nonNegativeInteger"/>
    </xsd:complexType>
</xsd:schema>
```
Appendix D

Pseudo Code for Control Data Flow Analysis

This chapter shows the pseudo code used by the control flow analysis of the MAPS framework.

D.1 SCFG Construction

Provide:

- \( V = (B, N) \) is the input LANCE CFG, where \( B \) is the node set and \( N \) is the edge set;
- Function \( \text{GetNextStatement}(s) \) returns the next IR statement after \( s \) in the same basic block, and \( \text{NULL} \) if \( s \) is at the end of the basic block;
- Function \( \text{GetFirstStatement}(b) \) returns the first IR statement of the basic block \( b \); and
- Function \( \text{GetLastStatement}(b) \) returns the last IR statement of the basic block \( b \).

/*G is the output statement control flow graph*/
\( G = (S, E) \), \( S = \emptyset \), \( E = \emptyset \)

Procedure ConstructStatementControlFlowGraph(V)
Begin
/*Analyze each basic block*/
For Each basic block \( b \) in \( B \) Do
  For Each IR statement \( s \) in \( b \) Do
    /*Get the next IR statement after \( s \)*/
    \( t = \text{GetNextStatement}(s) \)
    If \( s \) is not in \( S \) Then
      Insert \( s \) to \( S \)
    End If
    If \( t \) is not NULL And \( e = (s, t) \) is not in \( E \) Then
      Insert \( e \) to \( E \)
    End If
  End For
End For
D.2 SCDFG Construction

Provide:

- \( V = (S, E) \) is the input SCFG, where \( S \) is the node set and \( E \) is the edge set;
- Function \( \text{GetProducer}(s) \) returns the set of producer statements which write variables read by \( s \).

Procedure  ConstructStatementControlDataFlowGraph (\( V \) )
\( G = (S', E_c, E_d) \), is the output SCDFG.
Begin

\[
\begin{align*}
&/* SCDFG and SCFG have the same node set */ \quad S' = S \\
&/* The control edge set of the SCDFG is the same as the SCFG edge set */ \quad E_c = E \\
&\text{For Each statement } s' \text{ in } S' \text{ Do} \\
&\quad /* Get all producers of } s' */ \quad P = \text{GetProducer}(s') \\
&\quad \text{For Each statement } p \text{ in } P \text{ Do} \\
&\quad \quad \text{If IR statement } p \text{ is in } S' \text{ And} \\
&\quad \quad \quad \text{data edge } e_d = (p, s') \text{ does not exist in } E_d \text{ Then} \\
&\quad \quad \quad \text{Insert } e_d = (p, s') \text{ to } E_d \\
&\quad \text{End If} \\
&\quad \text{End For} \\
&\text{End For} \\
&\text{End}
\end{align*}
\]

D.3 Data Dependence Analysis

D.3.1 Static Data Analysis

Function  GetStaticProducer(s)
Begin

\[
\begin{align*}
&/* S is the output IR statement set */ \\
\end{align*}
\]
D.3. DATA DEPENDENCE ANALYSIS

S = φ
V = GetStatementUse(s)
For Each variable v in V Do
    D = GetDefines(v)
    For Each statement d in D Do
        If d is not in D Then
            Insert d to S
        End If
    End For
End For
Return S
End

D.3.2 Dynamic Analysis

Provide:

- T is the input trace;

- Function GetStatement(t) returns the accessing IR statement which is recorded in trace line t;

- Function GetTargetVariable(t) returns the variable which is accessed in trace line t;

- Function GetParentFunction(s) returns the parent function of the statement s; and

- Function GetCallSite(f) returns the call site of the function f.

/* This procedure is called once at the beginning of the analysis*/
Procedure ConstructDynamicDefUseTable(T)
    Use_d [s]: is the output dynamic Use table, which is indexed by IR statement, and each of its elements is the set of variables used by s.
    Def_d [v]: is the output dynamic Def table, which is indexed by variable, and each of its elements is the set of IR statements which define v.
Begin
    For Each memory trace line t in T Do
        s = GetStatement(t)
        v = GetTargetVariable(t)
        While true Do
            If t is a read access trace Then
                Insert v to Use_d [s]
            Else /* t must be a write access trace */
                Insert s to Def_d [v]
            End If
        f = GetParentFunction(s)
        If v is not locally defined in f And
            f is not the main function Then
            s = GetCallSite(f) /* Propagate the information to the call site of f */
        Else
            Break /* Jump out of the while loop */
    End For
End
End If
End While
End For
End

/* This function provides an interface to the dynamic dependence information */
Function GetDynamicProducer(s)
Begin
/* S is the output IR statement set */
S = φ
V = Use_d[s]
For Each variable v in V Do
  D = Def_d[v]
  For Each statement d in D Do
    If d is not in D Then
      Insert d to S
    End If
  End For
End For
Return S
End

D.4 WSCDFG Construction

Provide:

- \( G' = (S', E'_c, E'_d) \) is the input SCDFG;
- Function GetStatementWeight(s) returns the weight of the IR statement \( s \);
- Function GetControlEdgeWeight(\( e_c \)) returns the weight of the control edge \( e_c \);
- Function GetDataEdgeWeight(\( e_d \)) returns the weight of the data edge \( e_d \);

Procedure ConstructWeightedStatementControlDataFlowGraph(G')
/* G is the output weighted statement control data flow graph */
G = (S, E_c, E_d, W_s, W_c, W_d)
Begin
/* Copy nodes and edges from the input SCDFG */
S = S'
E_c = E'_c
E_d = E'_d
/* Annotate node weight */
For Each statement s in S Do
  W_s(s) = GetStatementWeight(s)
End For
/* Annotate control edge weight */
For Each control edge ec in Ec Do
  W_c(ec) = GetControlEdgeWeight(ec)
End For
End For
/* Annotate data edge weight */
For Each control edge ed in Ed Do
    W_d(ed) = GetDataEdgeWeight(ed)
End For

D.4.1 Node Weight Annotation

Provide:

- Function StmtExecutionCost(s) returns the overall execution cost of an IR statement s.

Function GetStatementWeight (s)
Begin
    w = StmtExecutionCost(s)
    Return w
End

D.4.2 Control Edge Weight Annotation

Provide:

- Function BBOverallExecutionCount(b) returns the overall execution count of basic block b;
- Function GetParentBB(s) returns the parent basic block of the statement s;
- Function GetFirstStatement(b) returns the first IR statement of the basic block b; and
- Function GetLastStatement(b) returns the last IR statement of the basic block b.

/* This procedure counts the occurrence of all inter basic block control flow in the complete trace T*/
Procedure CountInterBBControlEdge(T)
    Count_ce [e_c] is the globally inter-BB control flow occurrence count table.
Begin
    For Each function f in T Do
        CountControlEdgeInFunction(f) /* Defined below */
    End For
End

/* This procedure counts inter-BB control flow in one function */
Procedure CountControlEdgeInFunction(f)
Begin
    For Each trace line t of f Do

If t is a BB trace, which records the execution of BB b And there exists a BB trace line t' following t, which records the execution of BB b' Then

\[ s = \text{GetLastStatement}(b) \]
\[ s' = \text{GetFirstStatement}(b') \]
\[ e_c = (s, s') \]
\[ \text{Count}_{ce}[e_c] + = 1 \]

End If

End For

End

\/* This function returns the weight of the control edge \( e_c = (s, s') \) */
Function GetControlEdgeWeight(\( e_c \))
Begin
\[ b = \text{GetParentBB}(s) \]
\[ b' = \text{GetParentBB}(s') \]
If \( b \) and \( b' \) is the same basic block Then
\[ w = \text{BBOverallExecutionCount}(b) \]
Else
\[ w = \text{Count}_{ce}[e_c] \]
End If
Return \( w \)
End

D.4.3 Data Edge Weight Annotation

Provide:

- Function \( \text{GetParentBB}(s) \) returns the parent basic block of the statement \( s \);
- Function \( \text{BBOverallExecutionCount}(b) \) returns the overall execution count of basic block \( b \);
- Function \( \text{GetDependentVariable}(e_d) \) return the variable on which the dependence relationship represented by the data edge, \( e_d \), is established;
- Function \( \text{SizeOf}(v) \) returns the size of the variable \( v \) in bytes;
- Function \( \text{GetDefines}(v) \) is one the LANCE static data flow analysis functions, it returns the set of IR statements which define the variable \( v \); and
- \( \text{Def}_{d}[v] \): is the table constructed during the dynamic data dependence analysis. It is indexed by variables, and each element is the set of IR statements which define \( v \).

\/* This function returns the weight of the data edge \( e_d = (s, s') \) */
Function GetDataEdgeWeight(\( e_d \))
Begin
\[ b = \text{GetParentBB}(s) \]
\[ b' = \text{GetParentBB}(s') \]
\[ v = \text{GetDependentVariable}(e_d) \]
If $b$ and $b'$ is the same basic block Then /* Intra-BB data dependence */
$w = \text{SizeOf}(v) \times \text{BBOverallExecutionCount}(b)$
Else /* Inter-BB data dependence */
/* $D$ is the set of all IR statements which define $v$ */
$D = \text{GetDefines}(v) \cup \text{Def}_d[v]$
$B = \phi$
/* This loop convert an IR statement set $D$ to a BB set $B$ */
For Each statement $d$ in $D$ Do
	$b_d = \text{GetParentBB}(d)$
	Insert $b_d$ to B
End For
$\text{Count}_d = \text{The number of times that } b' \text{ is executed directly after } b$
$w = \text{SizeOf}(v) \times \text{Count}_d$
End If
Return $w$
End
Bibliography


[54] EU, “2PARMA project.”
http://www.2parma.eu.


[67] Compaan, “HotSpot Parallelizer.”


