Memory-aware platform description and framework for source-level embedded MPSoC software optimization

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Robert Pyka

Dortmund
2017

Tag der mündlichen Prfung:	16.03.2017
Dekan / Dekanin:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter / Gutachterinnen:	Prof. Dr. Peter Marwedel
	Prof. Dr. Jens Teubner

Acknowledgements

It is my pleasure to acknowledge the roles of several people who were instrumental for completion of this work.

First and foremost, I would like to thank my advisor Prof. Dr. Peter Marwedel for providing me with the opportunity to work on my PhD thesis in his group. I acknowledge his unwavering support over the last few years, his guidance and advice in so many discussions. He gave me the freedom to pursue my work in the most interesting direction, while contributing his vast knowledge and advice to shape this thesis into a valuable contribution to the research community. I have learned a lot from him, far beyond what can ever be concluded in a dissertation. I would also like to acknowledge all the helpful suggestions from my co-reviewer Prof. Dr. Jens Teubner and the committee members Prof. Dr. Heinrich Müller and Prof. Dr. Peter Buchholz.

Among all my former and current colleagues, I would like to mention in particular Felipe Klein, Florian Schmoll, Olivera Holzkamp and Daniel Cordes. They have been the most involved MACCv2 users and contributors. They spent countless hours on fruitful discussions and provided numerous improvement suggestions, bug reports and actual implementation contributions. Many personal thanks go to Olivera Holzkamp. She is the one who has always found the right words to keep me going and finalize this thesis. I am also indebted to Manish Verma for providing the initial inspiration to this work. A special thanks goes also to Rowena Worsley-Potthoff as well as to Michael Engel for their support, reviews and valuable comments and improvement suggestions to this dissertation.

This work has been supported in significant part by the EC Seventh Framework Program FP7 / IST-216224. I would like to express my gratitude to the members of the MNEMEE project. In particular, I am indebted to Stylianos Mamagkakis and Arindam Malik who coordinated this project and finally agreed on using the MACCv2 framework in this project, to Sander Stuijk for always supporting my work within the project and finally, to all the other participants who have lead the project to success and in this way made it a success story for MACCv2 as well.

Finally, many thanks goes to the students who I have been able to work with. I appreciate a lot all their cooperation and contribution to this work. In particular, thanks goes to Christoph Faßbach for his contribution of an exemplary optimization technique and to Selma Jabour and Frank Benneker for working on the Eclipse-IDE integration.

I owe my family my gratitude because they always supported and believed in me. I do so in particular to my parents, Barbara and Herbert. They have always been there for me. Above all, I would like to express my deepest thanks to my wife, Patricia, who gave me all her patience and comprehension for such a long time since the beginning of this adventure.

Düsseldorf, March 2017 Robert Pyka

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Contribution to Research Community . . . 6
1.3 Author's Contribution . . . 8
1.4 Structure Overview . . . 8

2 Related Work and Design Context **13**
2.1 Introduction . . . 13
2.2 System Modeling . . . 13
2.3 Frameworks . . . 18
2.4 Targeted Optimization Techniques . . . 19
2.5 Common-Object-Class Services . . . 21
2.6 ICD-C Framework . . . 22
2.7 MNEMEE Project . . . 26
2.7.1 Project Overview . . . 26
2.7.2 Toolflow Structure . . . 27
2.7.3 Project Outcome . . . 29

3 Design Specification **33**
3.1 Introduction . . . 33
3.2 Motivation . . . 34
3.3 System Description . . . 38
3.3.1 System-Model Structure . . . 41
3.3.2 Address Spaces . . . 49
3.3.3 Access Model . . . 50
3.3.4 Access Routing and Mapping . . . 51
3.4 Aspect Modeling . . . 54
3.4.1 Aspect Definition . . . 55
3.4.2 Value Composition . . . 57
3.5 Framework . . . 60
3.5.1 Processing-Step Integration . . . 62
3.5.2 Supplementary Framework Services . . . 68
3.6 Conclusion . . . 71

4 Implementation **75**
4.1 Introduction . . . 75
4.2 Framework Services . . . 76
4.2.1 Common Base Class . . . 76
4.2.2 Code Representation . . . 90

4.2.3 Runtime Environment 90
4.2.4 User-Interface Abstraction 93
4.2.5 Practical User-Interface Implementations 98
4.3 System Modeling 105
4.3.1 Practical Component and Channel Models 106
4.3.2 Common Aspect-Handler Examples 118
4.4 Processing Step Integration 121
4.4.1 Tool Implementation 121
4.4.2 Tool Specialization and Abstract Tools 123
4.4.3 Processing-Step Interaction and Configuration Options 124
4.4.4 Toolflow Construction 126
4.5 Conclusion 129

5 Evaluation Results **131**
5.1 Motivation 131
5.2 System Design Effort 132
5.3 Abstraction-Level Considerations 136
5.4 Target-Platform Representation Precision 137
5.5 Application Scenarios 143
5.5.1 Application in MNEMEE Project 144
5.5.2 Application in Teaching 155
5.5.3 Application for Present Techniques 155
5.6 Conclusion 157

6 Conclusion **161**
6.1 Summary 161
6.2 Future Work 162

Bibliography **165**

List of Figures **171**

1 Introduction

1.1 Context

Software development is a challenging task. Implementing algorithms is similar to the way humans communicate to each other, there is a myriad of ways to pass-on given abstract information. Assuming a given language, the knowledge of expected audience enables a more concise representation by reducing the amount of context information or using particular commonly known phrases. Choosing a particular way to express some information is typically guided by a set of limiting factors. Most likely, the primary factor is the time frame available for a given speech or the number of pages available in a publication. Finding the most informative representation within the given set of limitations is clearly a challenging *Optimization Problem*. Coming back to the software development process, straightforward relations to the natural language formulation exist. The target platform can be denoted as the audience. Detailed knowledge of the target-platform properties enables an implementation which reduces code overhead and utilizes all available hardware components. The motivation for implementing an algorithm in a particular way is analogous. The implemented task needs to be performed within a given time limit or a given energy budget. Finding sufficiently fast or energy saving implementation is again an optimization task. In contrast to natural languages, for computer programming languages, automated transformation and optimization approaches exist, which can be used to achieve a more appropriate representation while preserving the semantic notion. The work presented in this thesis provides the implementation foundation for a set of such optimization techniques which benefit especially from detailed target-platform knowledge.

Target platforms which closely interact with the surrounding environment often expose tight limiting factors. Primarily, manufacturing costs, reliability requirements and size and/or weight limits translate immediately into limited memory size, low computational power or a tight energy budget. Nevertheless, these *Embedded Systems* are becoming omnipresent and are required to perform increasingly complex tasks. These tasks vary across application domains. The automotive and avionic domains expose tasks in the area of engine control, position estimation, mitigation of injuries or human interaction devices. Typically, this domain exposes high reliability requirements in a harsh environment and especially in the automotive area also exposes high demand for cost efficient solutions. Besides the automotive domain, industrial applications are driven by embedded systems as well. This includes small-sized actuator and motor control at various power levels as well as large-scale distributed production process control networks. Finally, an interesting domain is the still emerging domain of mobile communication devices and corresponding networks. Especially hand-held devices expose all

kinds of limiting factors like weight, size, battery capacity, while requiring vast computational power enabling these devices to provide an appealing user experience and high data throughput. Common to all these applications is the focus directed towards the actual task to be performed, less towards the presence of a processor-based computer which is executing software. Actually, in many cases, a typical user will not be aware of using a computer. Nevertheless, a software-based approach helps to provide a flexible and eventually an in-field modifiable implementation. Knowledge of both the target application (which most likely will not change throughout the embedded system's life time) and the obscured computational part allow for choosing a well-tailored hardware platform. Especially, adding dedicated hardware blocks (i.e. communication processors, graphic processing units, etc.) and finding a reasonable trade-off between total system cost, necessary computational power and memory size are the knobs to be adjusted for a particular target application. Even though this non-exhaustive enumeration already spans a multidimensional design space, recent embedded systems designs add another dimension: The number of processing units. Many applications can be divided into concurrent subtasks. In terms of energy consumption it is advantageous to distribute these tasks to multiple, but slower, processing units, instead of executing them sequentially at high speed on a single CPU. As described by Rabaey et al. [1], this is primarily due to the power dissipation dependency on switching frequency and supply voltage in CMOS circuits. Due to the size limitations and interconnection performance requirements, multiprocessor embedded systems are implemented as *Multi Processor Systems on a Chip* (MPSoCs).

With respect to the initial example, in case of embedded software development, the challenge of implementing a particular application is comparable to having to give a talk on the same or similar topic to a variety of audience. On the positive side, for each particular talk, the audience is well-known. The straightforward approach would imply preparing the talk for each audience from scratch. Obviously, this is the least efficient approach. Nevertheless, initially in the case of embedded software development similar approaches were used. In those days, devices were programmed in assembler language. This allows for highly efficient implementations, but on the down side, the tractable complexity is limited. Furthermore, regarding current days' time-to-market requirements, prohibitively long development times would occur and the resulting software would still tend to be error-prone. In contrast to assembler languages where the main focus is directed towards the capabilities of the target processor, high-level programming languages provide an architecture-independent set of control-flow statements, a uniform way to express arithmetic and logic computations, a type-safe data storage management and typically a set of language-domain-specific features (i.e. message passing primitives in case of distributed systems programming languages). Using high-level programming languages moves developers' focus away from the actual target-platform peculiarities towards to the algorithm to be implemented.

The C programming language [2] has evolved to be the preferred language over recent years for embedded system software development. Primarily, the combination of aforementioned high-level language features, a small runtime-environment footprint and

the possibility to express efficient access to hardware components has rendered this language a good choice for embedded software development. The wide popularity of this language spurred development of corresponding compilation techniques. The C programming language is based on a sequentially imperative programing model. This model is well-suitable for single core architectures. Nevertheless, current embedded systems evolve towards multicore systems where concurrent execution of multiple processing threads is possible. Since there is no native representation within the C language which would allow implementing concurrently executing threads, typical approaches are based on function calls which encapsulate thread invocation, synchronization and data transfer.

Translation of C programs into assembler language representation and finally into an executable binary is typically performed via a tree pattern matching approach. Prior to this step, various *Source-Code Transformations* are going to be performed. In general, source-code transformations replace a sequence of statements or expressions with another one, which is semantically equivalent, but can be later translated into a more efficient binary representation in terms of runtime or energy consumption. The challenging task in supporting development of such source-code transformations is related to finding a solution for following requirements:

- **Provide a precise application-code representation which fully covers all C language features:** Especially for source-level transformations, the application-code representation must not perform any automated code simplification. All language patterns have to be preserved in the internal representation.

- **Provide a conversion path back to plain textual source-code representation:** Typically, source-level optimizations rely on a present compiler infrastructure. Once all optimizing transformations have been performed, the resulting source code has to be reconstructed to be fed into the compilation toolchain.

- **Provide an interface which allows for easy modification but ensures correct application-code syntax:** Performing syntactically-correct code modifications is non-trivial. The application-code representation has to provide an API which maps the code structure onto an object graph, where transformations of this graph relate to source-code transformations. This way application of only syntactically valid transformations can be ensured.

- **Provide support for data-flow and control-flow analysis:** Besides code transformations, optimization techniques need to analyze the code structure and corresponding data-flow paths in advance of taking optimization decisions. A graph-oriented representation is also beneficial for this task.

- **Provide tight integration with the system model:** Typically, optimization techniques relate application-code fragments and variations of those to particular system-model item. A uniform representation among both domains simplifies this task.

The work presented in this thesis relies on a well-suited source-code representation. Fortunately, the ICD-C [3] compiler framework could be used for this purpose. It offers an object-oriented interface to the application-code representation, conversion paths between plain-text and internal representation and several analysis and standard optimization steps. The challenging task in context of this thesis was related to the integration of the system model and the application-code representation into a uniform optimization technique foundation.

The actual benefit of such source-code transformations is dependent on the target-platform properties and the compiler used to generate the binary code representation. In straightforward approaches some trivial assumptions are made (i.e. less source code will result in a smaller binary code footprint, which in turn likely results in faster execution). Obviously, these assumptions do not always hold. Therefore, optimization techniques, which take target-platform properties into account, can achieve much better results. Superior to iterative approaches, where several transformations are evaluated on the target platform, and the one being most efficient is chosen, are optimization techniques which use model-based target-platform descriptions. These optimization techniques can immediately take the most suitable optimization decisions for a given platform. In particular, performing optimizations which take memory-subsystem properties into account can result in significant gains. Each instruction fetch and each data movement involves some memory accesses. Directing frequent accesses to fast or energy-efficient memories shows immediate benefits.

Among all the system-modeling languages developed in recent years, some incorporate a memory model. In almost all cases, they target generation of simulators, transformation to hardware description languages or code generator generation. Actually, none of them fits very well the needs of source-level optimizations. The first class of languages implements a behavioral model, which can be transformed into executable code on the host platform and thus supports simulation. Transformations to HDLs require primarily structural information. A fixed set of properties per component is required to interpret the description and translate it to lower-level HDL. Furthermore, a high level of detailed information is required right from the beginning of system specification, since generation of a complete and synthesizable hardware description is at focus. In contrast to this, for memory-aware optimizations, a quite abstract system model is usually sufficient. Finally, models for code generator generators focus on the internal structure and instruction set of processing units. In contrast to the application-code representation, the lack of a suitable system-model description led to the proposal of a novel approach which primarily targets source-level optimizations.

The major challenge in provision of a target-platform model suitable for source-level optimizations is related to finding a balanced representation, which provides sufficient information, while being as abstract as possible, to achieve low construction effort. In addition, in most cases more than one optimization technique is applied to a given source code. Ideally, each of them uses the same target-platform model to reduce the risk of contradicting optimization decisions. Basically, such an system-modeling approach has to fulfill following requirements:

- **Provide a combined locally scoped structural model with the typically required component-centric full-system property perspective:** There are in general two contradicting requirements: The system-model designer prefers a self-contained view of each system-model component, while the optimization technique developer prefers system-global property values which take all the particularities of a system into account. Especially by the means of proposed aspect handlers, the system-modeling approach provides a satisfactory solution for both requirements.

- **Provide a system-modeling approach feasible for modeling of a wide range of target-platform types:** Covering a wide range of target platforms implies either a vast set of predefined properties and system components, or a fully open set of properties. In the first case, significant implementation effort would be required, while still retaining the drawbacks of a closed system model. In the second case, optimization techniques need additional, often implicit, knowledge about the item names, and structural properties of the system description to be able to retrieve particular system property values. The approach proposed in this thesis uses component inheritance and the aforementioned aspect handlers to overcome these drawbacks.

- **Provide precise system property access at optimization technique runtime:** Typical full-system-simulation-based system property evaluation exposes a prohibitively prolonged response time. This renders optimization techniques which rely on such data infeasible for being applicable in compilers runs occurring in typical application development processes. The system-modeling approach proposed in this thesis provides a balanced implementation which is capable of provision of system property values at a precision level comparable to full-system-simulation-based approaches, while offering a fast, application-structure-independent and query-based interface. This allows for implementing optimization techniques being applicable in typical compiler runs.

The system-modeling approach proposed in this thesis fulfills aforementioned requirements and therefore solves the challenging task of provision of a versatile, precise and runtime-applicable system description.

Finally, supporting source-level optimization technique development is also related to provision of an implementation framework. The most challenging task in this domain is related to finding the right set of features being beneficial for development of source-level optimizations. Typical infrastructural properties of several optimization technique approaches in the domain of memory-aware source-level optimizations have been analyzed. This results in the following set of typical requirements related to the optimization technique framework:

- **Provide self-contained optimization technique representation:** A modular, self-contained optimization technique highly increases chances for successful reuse

in further optimization flows. The challenging task here is related to finding simple, uniform but sufficiently flexible interface definition.

- **Provide support for application in hierarchical optimization flows:** Optimization techniques are typically subdivided into fine-grained steps. These steps tend to consist at least of further analysis or transformation sub-steps. The framework proposed in this thesis provides a processing-step interface, which can be used uniformly across all hierarchy levels, while being able to keep track of tool dependencies across these levels.

- **Provide user-interaction facilities:** Optimization techniques may require user interaction while being applied to particular application code. Especially, in the case of current research work, where new ideas are evaluated, user interaction is beneficial. Within this framework a highly abstracted user-interface model has been developed, which allows for interactive optimization technique implementations which are independent of the current execution environment.

- **Provide data exchange and storage support:** Besides the actual application-code representation, multi-step optimization techniques need to pass additional meta-data between correlated processing steps. The challenging task in this context is related to provide an easily accessible, preferably native-language-based, method for data retention and data communication. The framework proposed in this thesis offers a sophisticated object model, which provides uniform and serializable data representation across the system-model description, the application-code representation and supplementary optimization technique data.

In general the framework solves the challenging task of bringing support and facilities for research-grade optimization technique development, while keeping focus on applicability of these techniques in production-grade compilation environments.

Concluding the introductory context overview, the work proposed in this thesis targets the domain of memory and target-platform-aware source-level optimization techniques. A well-balanced approach of application-code representation, target-platform model and supplementary services framework is proposed. This combination of these building blocks has shown its real-life applicability in the MNEMEE project.

1.2 Contribution to Research Community

Developing source-level optimization techniques for embedded MPSoC platforms is a challenging endeavor. The work presented in this thesis provides an infrastructure which facilitates this task.

There are two major areas of contribution. The first deals with platform representations. Applying optimizing code transformations typically requires knowledge of target-platform properties while taking optimization decisions, in particular since such optimizations are expected to improve performance either in terms of runtime or energy

consumption on a given target system. There are numerous ways to provide this information. Starting from implicit assumptions in the optimization technique implementation, over some lookup-table-based approaches, up to more generic optimization-technique-specific system descriptions. Even those system descriptions are limited to suit best one particular optimization technique implementation. The contribution in this area is provision of a uniform system-modeling approach which fits well the requirements of a wide range of memory-aware optimization techniques for embedded MPSoC platforms. The system model is located at a quite abstract system-modeling level, the processor-memory-switch level. This abstraction level fits well requirements of memory-aware optimizations which typically need to acquire a system overview first (i.e. type and count of memories) and later on need detailed memory-component-related information (i.e. per-access energy consumption). This system-modeling approach features a unique method to calculate on-the-fly system-wide properties based on local definitions within components and interconnections. This allows for a very modular approach, while still providing precise system-level properties to optimization techniques. A growing set of components and channels, which can be easily combined and adjusted, enables low effort modeling of various system-model descriptions. The object-oriented system-modeling approach allows for well-structured component and channel property representation. Especially, inheritance relations are a valuable way of finding a particular class of system-model components or acquire class-specific details. The system-modeling approach presented here has found community acceptance at the LCTES'10 conference. Refer to Pyka et al. [4].

In the second area of contribution an optimization technique framework is presented. Proposing a theoretical system-modeling approach in isolation would be of limited benefit once an actual optimization technique has to be implemented. Therefore, the system-modeling approach presented in this thesis has been embedded into an optimization technique framework. The unique combination of an infrastructure which supports optimization technique cooperation and data exchange, accompanied by the aforementioned system-modeling approach and a detailed application-code representation provides a solid foundation for rapid optimization technique development. Encapsulating optimization, analysis and code transformation techniques into so-called tools which expose uniform invocation interfaces is one of the key properties of this framework. This simplifies invocation of a variety of commonly occurring tasks. As can be observed later on for the MNEMEE toolflow in Section 5.5.1.3, especially construction of toolflows of such self-contained building blocks becomes available at significantly reduced effort. Furthermore, the framework provides commonly-used services to optimization techniques. This includes data retention, source-code and system-model annotation techniques, a common-object-class model and corresponding reflection methods and a user-interface abstraction layer.

The proposed MACCv2 framework, including the system-modeling approach, has been successfully used as the core integration service in the MNEMEE project. The project's resulting optimization toolflow translates an initially sequential application code into a parallel representation. The primary goal targets a more energy-efficient utilization of

available processing resources and related memories. Furthermore, the MACCv2 framework has been used in teaching activities at ALaRI [5]. The object placement computed in the scratchpad-memory optimization could be interactively presented to students in hands-on sessions. The approach supports effortless tweaking of target-platform properties (i.e. memory sizes or energy-model values) and observing the resulting optimization decisions.

Concluding the contribution, the approaches presented in this thesis have been able to show their added value in a real-life project, teaching, related PhD theses and subsequent research work.

1.3 Author's Contribution

Summarizing author's contribution to the work presented in this thesis two areas of contribution are considered separately. Regarding conceptual work, models and approaches Table 1.1 shows an enumeration of author's contribution to particular topics of this thesis. Except for the source-code representation, the author is the only contributor.

Table 1.2 shows an overview of author's contribution to the implementation topics relevant in this thesis. Here, full contribution has been limited to the core MACCv2 framework components, while contribution to remaining implementation topics, which have been primarily performed in the context of MNEMEE project, has been limited to provision of implementation support and starting point templates.

1.4 Structure Overview

This thesis is structured into six chapters. Following this introductory chapter, references to related work and design context are given next.

Chapter 3 presents models and design specifications regarding the major contributions of this thesis. Preceded by an introduction and motivation, the proposed system-modeling approach is presented first. Key properties are enumerated and a structural overview is presented to the reader. The next section presents the aspect-modeling approach. Aspects in terms of this thesis are arbitrary values which will be computed on-the-fly according to the system-model structure, while still being accessible in a database-like style. The last section focuses on the framework. A structural overview is presented, followed by a description of major framework services. In particular these are:

- Interfaces to the system-model representation.
- Target-platform property computation.
- Processing-step representation.
- A set of supplementary runtime services.

A final conclusion summarizes the proposed design specification.

Topic	Aprox. contrib. %	Remarks
System description	100%	
Access enumeration	100%	
Aspect handlers	100%	
Framework basic services	100%	
Framework tool model	100%	
Framework user interface abstraction	100%	
Framework runtime env. and library handling	100%	
Exemplary opt. technique in Section 5.5.3	10%	Developed in context of diploma thesis advised by author.
ICD-C code representation	5%	Product of ICD e.V. The author added common object class and serialization support.

Table 1.1: Author's contribution to concepts and models.

Chapter 4 takes a close look at the actual implementation and corresponding examples. The set of general framework services around a common-object-class model is presented first. In particular these are:

- Base-class services.
- Code representation.
- Runtime environment.
- User interfaces.

Based on this foundation, implementation details of the system-modeling approach and processing-step representation are described. As appropriate, sections are accompanied by practical example of presented items. A final conclusion summarizes the proposed implementation.

Within Chapter 5, the system-modeling approach and framework evaluation results are presented. Evaluation has been performed according to a set of goals initially motivated: First, the effort required to provide a system model is discussed. Especially, the possibility to control the effort by choosing the appropriate level of detail is a valuable property of this approach. Next, a subsection focuses on the abstraction level chosen

for this system-modeling approach. Here again, a trade-off between level of detail and accessibility exists. The processor-memory-switch level has been identified as the most appropriate abstraction level for the needs of memory-aware optimization techniques. Finally, the precision of this approach is discussed. Since the access method significantly differs from known methods (i.e. simulation-based approaches) a comparison is performed, showing this approach as being capable of providing similarly precise values in a fraction of time compared to state-of-the-art methods. Hereafter, the applicability of this approach to real-life problems is evaluated. There, the benefits for the MNEMEE project are presented, applicability in courses has been shown, and finally the effort and benefits of porting an already existing optimization technique to this framework is discussed.

This thesis is concluded in Chapter 6. The proposed system-modeling approach and framework qualifies as a viable way for rapid development of memory-aware source-level optimization techniques. Towards the end of this chapter, limitations and possible future extensions are discussed.

Topic	Aprox. contrib. %	Remarks
MACCv2 core library	100%	
Basic source-code optimizations tool	100%	
Compiler/build tools library	30%	Developed by other MNEMEE participants. The author provided templates and implementation support.
Run/profiling tools library	20%	Developed by other MNEMEE participants. The author provided templates and implementation support.
Generic command-line executables	100%	
Host platform template	100%	
MPARM platform template	40%	Structural model provided by the author. Energy and latency handlers provided by Klein [4].
COMET platform template	10%	Developed in the context of the MNEMEE project.
MSC8144 platform template	10%	Developed in the context of the MNEMEE project.
UI texmode	100%	
UI graphical (Qt)	100%	
UI for Eclipse-IDE	10%	Developed in the context of a diploma thesis advised by author.
MNEMEE toolflow integration	60%	The author provided a top-level tool and templates for tool interfaces classes.
MNEMEE optimization steps	5%	Developed by other MNEMEE participants. The author provided implementation support.
MNEMEE toolflow frontend UI	100%	
Application in teaching	5%	The author provided contributions to the graphical user interface.

Table 1.2: Author's implementation-related contribution.

2 Related Work and Design Context

2.1 Introduction

The work presented in this thesis incorporates several disjoint aspects which have relations to previously performed work in the research community. First of all, the proposed system-modeling approach is put into relation to previous research. In particular, the general classification according to the abstraction level is of prominent importance. Since besides the system-modeling approach a fullfledged framework supporting development of memory-aware optimization techniques is proposed, corresponding related work is analyzed next.

Next, an overview of possible application domains for this framework and system-modeling approach is given. This is done by providing a concise enumeration of targeted optimization techniques for which this framework is expected to be beneficial.

Later on, implementation techniques used in this framework are referenced and shortly introduced. In particular, these are the various common-base-class services and the application-code representation ICD-C.

Finally, the MNEMEE project is presented in the last section of this chapter. The work presented in this thesis has been primarily carried out in the context of this project. The contribution presented in this thesis has been used as the integrating foundation in MNEMEE.

2.2 System Modeling

This thesis proposes a framework for fast and effort-efficient development of memory-aware optimization techniques. One of the highlighted features of this framework is the target-system description. Using a common system description across various optimization techniques, integrated by the means of this framework, allows for well-coordinated

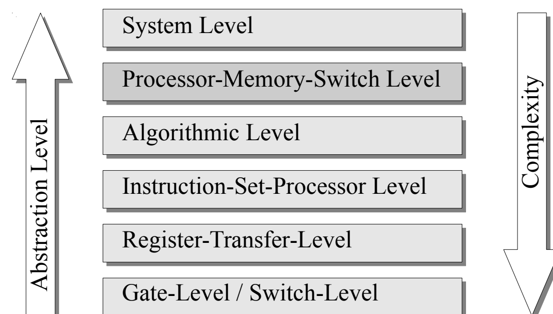


Figure 2.1: System-modeling levels

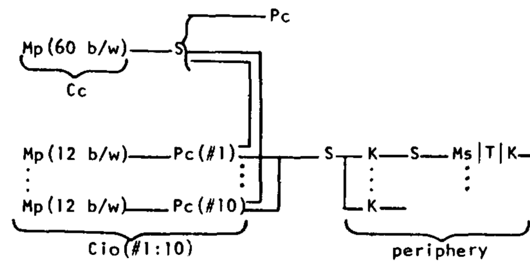


Figure 2.2: Exemplary Bell/Newell PMS representation of a CDC 6600 system. [7]

processing and higher optimization gains. Furthermore, for reasonably fast executing optimization techniques, target-platform properties need to be accessible instantaneously without prolonged full-system simulation. Such a target-system description requires a definition using a suitable system-modeling language. According to the classification of system-modeling languages along the abstraction level as shown in Figure 2.1, the description language presented in this thesis can be assigned to the PMS level. Processor-Memory-Switch models were introduced by Bell and Newell [6]. They describe the system at an abstract level where the main building blocks are the system components (i.e. processors and memories) and the interconnection between them. The primary application for PMS models was provision of a description of structural properties of computer systems and the interconnections between them. Originally, PMS models define seven item types:

- Processors
- Memories
- Switches
- Links
- Data operation
- Transducers
- Controllers

In a graphical representation, instances of these items were connected via lines to each other to represent a relation between them. Annotations to these items provide further classification and higher level of detail, if required. These optional annotations allow for subsequent refinement of system models. Figure 2.2 presented by Bell and Newell depicts such an exemplary structure.

Nowadays, improvements in chip manufacturing enable much higher integration, resulting in on-chip structures which resemble full computer networks at the time of introduction of PMS models. Refer to Jantsch and Tenhunen [8] for a description of further

properties of such Networks-on-Chip (NoCs). Nevertheless, the need for a representation of such systems still persists. Therefore, the corresponding PMS abstraction level has preserved its value. The system-modeling approach presented in this thesis takes advantage of the PMS abstraction level. Especially, for the domain of memory-aware optimization techniques a balanced system description is required. It is expected to expose sufficient target-platform properties, without being difficult to handle, due to excessive level of detail. Please refer to Section 3.2 for a more in-depth motivation.

Subsequently, various architecture description languages (ADLs) have been developed in recent years. One comprehensive approach to classify them is by the main application target. A first application target is the automatic transformation into a hardware description language. This helps in automating hardware development as well as in shortening the development cycle, where precise simulators can be derived in advance from a HDL representation. A second main application target is the automatic generation of development tools like compilers, assemblers and linkers. Both application targets impose almost disjoint requirements on the ADL. On the one hand, hardware description requires precise structural information, on the other hand, automatic tool generation requires a behavioral and semantic description of the system. In the domain of memory-aware optimization techniques, focus has been put on the actual optimization technique. In most cases an ad-hoc approach has been used which provides the required parameters in an optimization-technique-specific way. Therefore, this thesis proposes a new systematic approach in the domain of memory-aware optimization techniques, which is based on a common target-platform model. Since memory optimizations typically require abstract structural properties enriched by individual semantic information, several ADLs used in the domain of HDL construction and development tools generation are relevant to this thesis. Even though they provide a valuable foundation, none of them offers the appropriate scope to be directly usable within the MACCv2 framework.

LISA [9] is an ADL which targets primarily automatic generation of application-specific hardware and corresponding simulators and low-level tools. Primary architectural targets are signal processing and generic irregular single processor architectures. The language has been extended later towards automatic compiler generation. To accomplish this task, an additional semantic instruction set model has been added by Ceng et al. [10]. Since the main target of LISA is the cycle-true description of a digital signal processor, neither a sophisticated system model nor detailed memory model exists. The timing model integrated into LISA focuses on the specification of the pipeline behavior. No energy model is incorporated into LISA. Muhammed et al. [11] describe a subsequent extension to LISA, which provides a resource model. Still, this resource model is quite processor-centric and targets a simple enumeration of memories and their sizes and few other resource types.

ArchC [12] is another currently available ADL. ArchC was designed to support processor architecture description. While the language has evolved, also the possibility to design memory hierarchies has been added. Similar to LISA, ArchC covers the structural and behavioral view of a system model. Since ArchC is based on the SystemC language, which provides extensions to C++ for description of timing and concurrency, ArchC

models are described in C++ code as well. On the positive side, this allows for the description of a large variety of different systems, on the negative side, once it comes to other tasks than simulator generation, it is really hard to extract any semantic meaning from such a model. The SystemC language offers all the expression possibilities of the C++ language. This is perfect for simulation, since this allows for efficient implementation of compiled-simulation simulators, which is the fastest simulation method available. Recent work by Schürmans et al. [13] shows even the applicability of SystemC models for power estimation in full-system simulation. But even the task of statically extracting timing information out of such a model without imposing any restrictions to the modeling style is an almost intractable problem. Closely related to ArchC is PDesigner [14]. Basically, it is a graphical editor which can be used for an intuitive, component-based development of ArchC system models.

Specialized system descriptions targeting mapping of applications on MPSoCs are another group of system-modeling languages. In this context, references to the SpecC language developed by Gajski et al. [15], the ADL used in DAEDALUS [16] or the hardware platform description in the CIC-based (common-intermediate-code-based) retargetable parallel programming framework for MPSoC by Kwon et al. [17] are provided. Finally, the Distributed Operation Layer (DOL), as presented by Thiele et al. [18], can be considered to be within this group as well. In general, these system descriptions are from the structural point of view similar to the one presented in this thesis. Especially, SpecC exposes a similar channel- and component-based structure. The major difference to the work presented in this thesis is the application centric platform view of these modeling languages. On the one hand, platform specifications as occurring in DOL focus on the communication structures important to the target application. In the first place, such a model is not going to resemble the actual physical target-platform structure, but will contain abstract channels which represent end-to-end links between processing units. On the other hand, approaches like SpecC focus on target-platform hardware structure adaptation for a particular application. Trying to use these modeling languages in the reverse scenario for application-code modification on a fixed platform would require various optimization-technique-specific annotations to the system model. These approaches would suffer from the fact that changes to the properties of one component need to be annotated in several places of the system description. The energy consumption of a memory component would be an example for such an annotation. In contrast to this approach, a change of such a value would also affect the per-access energy values annotated to processing elements. In general, this will require user interaction and in-depth system knowledge to precompute these values.

EXPRESSION [19] was developed in the late 90's. It aims primarily at automatic generation of software development tools. The motivation for this language was faster design space exploration (DSE) on a single processor of the SoC. To accomplish this task, EXPRESSION describes the system in a structural and behavioral way. In contrast to previous languages, EXPRESSION offers an explicit memory model. There is only a fixed set of parameters which can be used to describe properties of memories available

to the processor. On the downside of this ADL, there is no method to convert the description into a HDL for automatic generation of processor hardware.

Regarding design space exploration, an interesting approach is presented by Diewald et al. [20]. The proposed Exploration Meta-Model (EMM) focuses on coordination of design space explorations in the context of Model-Driven Development (MDD). Since the MACCv2 system description can be used for fast exploration of target-platform variations, but models for guiding these variations are out of the scope of the MACCv2 framework, both approaches perform complementary roles. Therefore, DSE-related optimization techniques are expected to benefit from a combined application of both approaches.

A slightly different application scenario is specified for the TDL [21] language. The primary goal of this language is to support development of retargetable post-pass optimizations at assembly level. This is what comes closest to the approach proposed in this thesis. TDL includes a structural description of resources present in the system. This includes memories and corresponding cache hierarchies. Furthermore, a behavioral description of the instruction set is the second key part of the TDL language. Nevertheless, there are significant differences to the proposed ADL. The semantical information in the resources section does not allow for modeling structural dependencies between memories. Furthermore, only single processor-based memory hierarchies can be described.

A related approach for platform description is presented by Kessler et al. [22]. Their XPDL language targets a similar application scenario as the one presented in this thesis. In general, system-level target-platform properties can be retrieved via a C++-based API. These properties are stored precomputed within the XML system description. If some properties are missing, a microbenchmarking-based profiling step is triggered to estimate these values. In contrast to the approach presented in this thesis, which calculates these values on-the-fly solely from locally scoped properties attached to each component, the XPDL approach operates similar to typical table-lookup approaches. Frequent platform modifications require time-consuming reprofiling of these values.

Finally, a vast variety of other ADLs exists. As examples, references to hardware-related ADLs like IP-XACT [23], nML [24], ISDL [25] and MIMOLA [26] are given. All of them do not focus on development of memory-hierarchy-aware source-code transformations. The common goals are simulation, HDL extraction or compiler generation. None of them satisfies all the requirements imposed once source-level optimizations for multicore platforms are targeted. In a broader scope, the set of software-related ADLs also has to be taken into account. Examples in this class of ADLs are Wright [27] or Darwin [28]. In contrast to this approach, these ADLs concentrate on description of the application architecture instead of the platform the application is being executed on.

Concluding the overview of system-modeling-related work, several system-modeling approaches have been identified. In general they can be classified into two groups. The first one, which focuses on behavioral models, targets construction of full-system simulators. The second one, which focuses primarily on structural properties of processing units targets compiler and design tool development automation. Actually, none of them takes

requirements of memory-aware source-to-source transformations into account. Furthermore, the preferable abstraction level, and therefore the level of detail, highly depends on intended application scenario. The PMS abstraction level offers the best suitable balance between modeling complexity and level of detail.

2.3 Frameworks

The system-modeling language presented in this thesis is embedded into a framework which provides a foundation for the development of various optimization techniques. Several of previously-referred system-description languages have a similar relation to a framework in their specific domain. Most noticeable are LISA [9], DEADALUS [16] or MIMOLA [26]. The MACCv2 framework proposed in this thesis focuses on source-code transformations which strive to achieve an optimized representation of a target application in terms of energy consumption or runtime. A relative in this domain is the LLVM-framework [29]. LLVM provides an infrastructure targeting compiler implementation for various programming languages. Typically, several subprojects concentrate on parsing of source code (i.e. Clang, Dragonegg), debugging (LLDB), providing a set of core libraries which implement transformation, optimization and lowering steps which translate the source-code representation into an assembler-like intermediate representation. This low-level representation can be later on translated into target-platform executables regardless of the programming language used up-front. To some extent both frameworks, LLVM and MACCv2, perform the same task, namely, performing code transformations. Nevertheless, major differences exist: LLVM focuses on code compilation, and provides corresponding optimization techniques, while MACCv2 focuses on source-to-source transformations which target efficiency improvement in terms of energy consumption or runtime via optimized memory-access patterns. In contrast to this framework, where a fullfledged model of the target-platform memory subsystem exists, the LLVM framework offers a set of target-platform properties, which are relevant for code generation and therefore are focused towards processor unit description. An example in this area is the work presented by Grech et al. [30]. They propose a processor centric application-code energy-consumption estimation. The estimation is based on a per-instruction energy-consumption model which does not model any further platform properties. In general, LLVM can be considered to have the same scope as the ICD-C compiler framework being a component integrated into the work presented here.

A framework which also targets memory-aware compilation has been presented by Verma et al. [31]. It consists of compilation and simulation parts which operate on a common energy data base. Both parts are focused towards an ARM7 architecture. A self-implemented compiler and corresponding analysis steps allow for implementation of dedicated and deeply integrated optimization techniques. Nevertheless, this approach comes to its limit, once retargetability becomes frequent and more important. The implementation effort for setting up the specific compiler and simulator infrastructure for each new target platform is prohibitively high. This has been the initial motivation for provision of a framework which is also capable of exploiting memory hierarchy properties

in optimization techniques, but does not have such a static structure and utilizes existing tools and compilers to achieve a similar compilation and evaluation loop.

A similar approach of combining a fixed set of tools for MPSoC-related software optimization is provided by Iosifidis et al. [32]. The framework, called MPMH, consists of three dedicated transformation steps targeting distinct aspects of application-code optimization. First, the application code is transformed into a parallel representation according to some annotations passed along the actual application code. The second step takes care of appropriate static data allocation and finally the last one does dynamic data structure transformations. A simple, list-based, platform description provides the enumeration of target-platform properties guiding these transformation steps. In contrast to the work presented in this thesis, MPMH framework does not focus on providing a foundation for development of memory-aware optimizations. Therefore, even though the targeted application scenario is the same, the scope is completely different. In a comparison, MPMH framework would rather relate to a particular instance of combined optimization techniques implemented using the MACCv2 framework. This relation can be observed in the evaluation section, where the application of MACCv2 framework as foundation for the MNEMEE toolflow [33] is presented. In particular, the MNEMEE toolflow incorporates wrappers for MPMH optimization steps. Hence these steps have been used as individual processing step within this optimization flow.

Concluding the framework-oriented related work, efficient implementations of memory-aware optimization techniques are best founded on a common framework. Well-known frameworks focus either on hardware extraction or compilation. Especially, the last type of frameworks contributes a variety of source-code transformations. Nevertheless, they do not take memory-subsystem properties into account. The work presented in this thesis, provides a unique combination of an abstract system model and a corresponding framework which targets the domain of memory-aware source-level optimizations on embedded multicore systems.

2.4 Targeted Optimization Techniques

Designing embedded systems always has the abstract goal of finding the most efficient implementation. Unfortunately, there are contradicting limitations like cost, size and timing constraints which reduce this goal to finding some balanced implementation among these constraints. Initially, both hardware and software were freely adopted to the target scenario, which results in high development effort. Once chip manufacturing costs hit millions of dollar per mask and short time-to-market times were demanded, platform-based design became the predominant design approach. Sangiovanni-Vincentelli and Martin [34] provide an overview of platform-based design aspects. Along the transition towards platform-based design, source-level code optimizations become more and more relevant. In particular, platform definitions are accompanied by a mature compilation toolset. Often the rapid time-to-market inhibits development of platform instance optimized compilers. Therefore, target-platform-aware source-to-source optimization are the

preferred method to achieve an efficient mapping of application code onto a particular target-platform instance.

Performing source-level optimizations has a long history. In the late 70s, Loveman [35] proposed various loop transformation methods targeting high-level programming languages. In this work it has already been recognized that applying source transformations leads to a target-platform-dependent benefit. Loop transformations have been important optimization methods for years. Sophisticated approaches which utilize polyhedral modeling have been presented by Falk and Marwedel [36]. Especially, benefits of high-level representations and source-to-source transformations have been analyzed and promoted in their work.

Regarded at a broader scope, loop optimizations can be considered as a subset of optimization approaches which gain their benefits due to reorganized memory-access sequences. Even though typically not at the same optimization point, both runtime reduction or energy-consumption reduction can nevertheless be achieved this way. Especially, energy-consumption reduction under the constraint of keeping runtime within required bounds is a prevalent topic in embedded system design and particularly in embedded software development. There are numerous aspects to be considered which could, depending on target-architecture properties, lead to desired energy-consumption reduction. On the one side, there are optimizations, like the aforementioned loop optimizations, which lead to energy savings due to reduced memory-access counts, either due to reduced code size, reduced data accesses counts or beneficial cache access sequences. On the other side, optimized placement of a particular code or data item has also a significant effect on energy consumption and runtime. Typically, embedded systems expose complex memory hierarchies with several memories differing in size, access latency, energy consumption or accessibility. More details on fundamental computer architecture properties are provided by Hennessy and Patterson [37] and a view more focused toward embedded systems is given by Marwedel [38].

Finding an efficient and feasible placement of application code and data has been a challenging task for years and still engages the research community. Panda et al. [39] provide a survey on a broad set of optimization techniques, while Wolf and Kandemir [40] present a more software-focused view on possible approaches. For practical examples references to the work done by Lai et al. [41] in the data-flow-oriented application domain, as well as to the work done by Wehmeyer and Marwedel [42] in the control-flow-oriented application domain are given. In the last one in particular, a closer look at the energy-consumption aspects of various memory types is taken and beneficial optimizations for these types are suggested.

Improvements in integrated circuit manufacturing technology allow for development of complex systems on a chip. These systems typically expose multiple processing units as well as several on-chip memories. These components are connected via multiple hierarchy levels of buses or even more complex interconnection channels. These complex Multiprocessor Systems on a Chip (MPSoCs) still need efficient mapping of target applications on a particular platform instance. Single core techniques including data placement and loop transformations can be adapted to multicore platforms. Verma and Marwedel [43]

show particular approaches in this area. Even though presence of multiple cores already adds further complexity to the memory-access optimization step, automated distribution of application code to these processing units adds an additional optimization dimension to the application-code mapping problem. Complexity grows by the variation in types of processing units included in a system. Homogeneous platforms have all processing units of the same type, followed by platforms where two types of processing units are included, both expose the same ABI but their internal structure implies operation at different speed and differing energy-consumption level. Kumar et al. [44] provided some considerations on these so-called big.LITTLE architectures. Continuing mapping problem complexity considerations along processing-unit variation, mapping of applications on systems having each processing of different type, can be considered the most generic, but also most complex approach to accomplish such mapping task. Techniques which perform application-code distribution across multiple processing units have been proposed by Cordes et al. [45] and later on accompanied by techniques presented by Jovanovic et al. [46], which actually map such distributed application code on particular processing units including consideration of efficient memory-hierarchy utilization.

Concluding this fraction of approaches targeting optimization of embedded systems software, there are a few recurring prerequisites needed for successful performing of such optimizing code transformations:

- A notion of target-platform structure and available component and interconnection properties.
- An optimization-decision guiding set of performance values. These may be execution times, energy-consumption hints, etc.
- An easy to modify, precise and usable for plain source-code reconstruction application-code representation.

The work proposed in this thesis aims at the provision of a common foundation for optimization techniques in the area described in this section. Therefore, among others, concluded optimization technique prerequisites have been addressed and corresponding models and implementations are provided. In fact, previously referred approaches by Jovanovic et al. [46] and by Cordes et al. [47], including the corresponding PhD theses [48], [49] benefit from this work. They use the MACCv2 framework and corresponding system models as a foundation for their optimization and transformation technique implementations.

2.5 Common-Object-Class Services

The framework services and approaches have been implemented in the C++ programming language [50]. The C++ language is commonly used in the area of memory-aware optimization techniques. This is often due to the availability of required libraries and tools (few examples are: lp_solve API library [51], LLVM pass implementation interface [52] or the ICD-C framework [3]). Further, nowadays the decision to implement an

upcoming optimization technique in C++ is historically motivated, since previous work has been implemented this way, such an implementation promises the least effort and a fast result. Therefore, this observation has been the natural motivation to contribute this work as a C++-based framework. Even though C++ is an expressive and powerful language, especially once it comes to support runtime linking and modularity it gets to its limits. In the case the application image is constructed of independent libraries of code at runtime, the code needs to become self-aware. In particular, it needs to have a notion of class names and their relations, have an insight into the class structure at runtime and provide some means of building object graphs of at compile-time-unknown object-classes. In the same context persistent data retention is based on the same object structure. These object structures need to be capable of being persistently stored and restored in some later runtime context. These requirements have been recognized to exist in the context of memory-aware optimization technique development as well. Therefore, the MACCv2 framework presented in this thesis has been designed to implement a set of services which provide reflection, common-object-class representation, object factories and serialization. Initially, a broad scoped presentation of design patterns targeting these aspects was given by Gamma et al. [53]. Especially, there are numerous approaches for object serialization. Maeda [54] gives an overview of these approaches. Besides C++, other languages have native support for common object model and corresponding serialization support. The most prominent is Java [55].

Modularity and runtime linking has another challenging point. Since the complete application context is not known a priori, typically the scope of an implementation is at best limited to library boundaries. Interaction between objects of different libraries needs dedicated support. Basically, there are two aspects to consider. First, objects typically hold references to other objects. Since the referencing set of objects is not known, some reference counting and pointer tracing is needed to ensure a consistent data structure state. Henney [56] provides a survey on approaches found in this area. The second aspect targets object notification. Since object state changes are of interest to other objects, some notification mechanism is needed. Gamma et al. [53] introduced the Observer design pattern, where current days programming languages like Java provide corresponding implementations [57].

2.6 ICD-C Framework

The MACCv2 framework presented in this thesis focuses on the provision of a foundation for the development of source-level optimization techniques for MPSoCs. Since a typical application scenario of MPSoC devices targets embedded applications, a typical application code to be optimized is implemented in the C programming language. Therefore, addressing this scenario requires an application-code representation for the C programming language. The most straightforward application-code representation operates on the plain textual representation. Although this is the most universal way to represent code, it offers least syntactic details. Also, performing changes is equivalent to editing the text, which happens without any syntactical correctness guarantees.

Therefore, except for some very basic cases, where text-based source-code modifications could be assumed to be the preferred way, any reasonable optimization or transformation technique needs an intermediate representation (IR) to perform the code modifications in a well-defined, and syntactically-correct way. Performing source-level code transformation via an intermediate representation requires translation of such a textual representation into the IR and, once all optimizations and transformations have been performed, back to a textual representation. To achieve these translations, source-level intermediate representations typically parse the textual representation and try to construct out of this token stream an abstract syntax tree according to expected programming language. Since performing sophisticated source-level optimizations is still an emerging technique, typical application scenarios for abstract-syntax-tree-based (AST-based) code representations target compiler development. There, the reverse step, reconstruction of a textual representation out of an AST, is not required. Therefore, most intermediate representations focus only on a semantically correct representation, but allow or even perform intentionally, simplifications, which prevent reconstruction of the original application code.

The ICD-C [3] intermediate representation used in MACCv2 framework has been designed for both application scenarios: compiler development as well as source-level optimizations. Therefore, it supports such reconstruction of source code out of its intermediate representation. Since the precise application-code representation has been at focus, the complete set of ANSI-C language constructs is covered by this intermediate representation. Some additional features (i.e. some GCC extensions or source-comment representation) are also provided, allowing application even for a specialized domain of code (i.e. automotive code). Besides the intermediate representation several standard and architecture-independent analysis, optimization and transformation techniques are provided within ICD-C.

In the context of this framework, the previously available ICD-C intermediate representation has been slightly extended. Basically, the major goal was to make ICD-C benefit from the advantages of a common-object-class model. Therefore, the ICD-C intermediate representation has been modified to fit the approach presented in Section 4.2.1. Due to this integration, a common foundation exists across the system-modeling implementation, the application-code representation and the processing-step integration approach provided within this framework. Such a cross domain foundation is a subtle, but very beneficial property of MACCv2 framework. Expression of optimization-technique-dependent cross-object relations and annotations is possible with only minimal effort for the optimization technique developer.

A closer look at the properties of ICD-C application-code representation shows an overall structure as depicted in Figure 2.3. At top level the representation is enclosed by a single object which describes a self-contained IR. Each intermediate representation consists of a set of compilation units and a set of global application symbols (i.e. global variables). A compilation unit basically relates to a single source-code file including all header files. At the next level the structure follows the grammatical structure of the C programming language. Therefore, compilation units consist of function definitions and

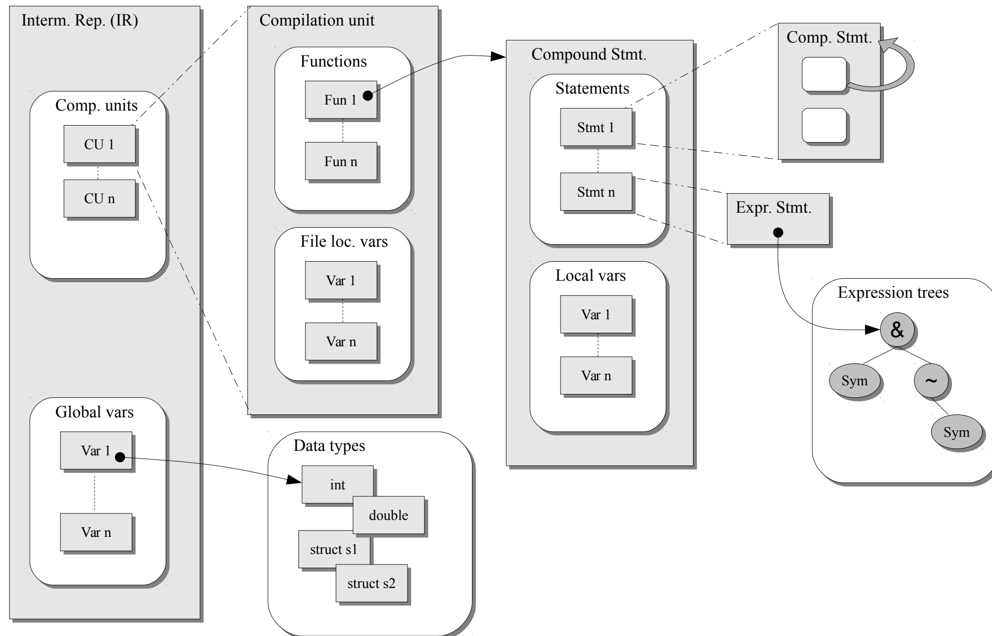


Figure 2.3: ICD-C IR structure overview.

file-scoped symbol declarations. Function definitions are basically structured in hierarchical statement lists. All language-relevant statement types are provided. The typical nested hierarchical code structure is represented via compound statements, which themselves contain a statement list. Other types represent loops, conditions or expressions, including the most commonly appearing assignment expressions. Especially, the expression statements are interesting in this structural overview because they provide a link to the expression representation. Expressions are represented as a tree. Except for the representation of symbols or constant values, each expression consists of one or more sub expressions. According to the evaluation order, such expression elements form expression trees.

Besides the abstract syntax tree, also semantical information is provided within this application-code representation. First of all, data type representation is provided for each symbol and each expression. Herein, relations like type modifications, storage class assignments and, if applicable, complex data type membership relations are expressed. Further semantical information is deduced from on-the-fly analyses. This includes among others, function-wide data-flow analysis and control-flow analysis. Based on this intermediate representation a set of sophisticated analyses is provided. This includes loop bound analysis or an alias analysis. For further details on the properties of the ICD-C code representation, please refer to [3].

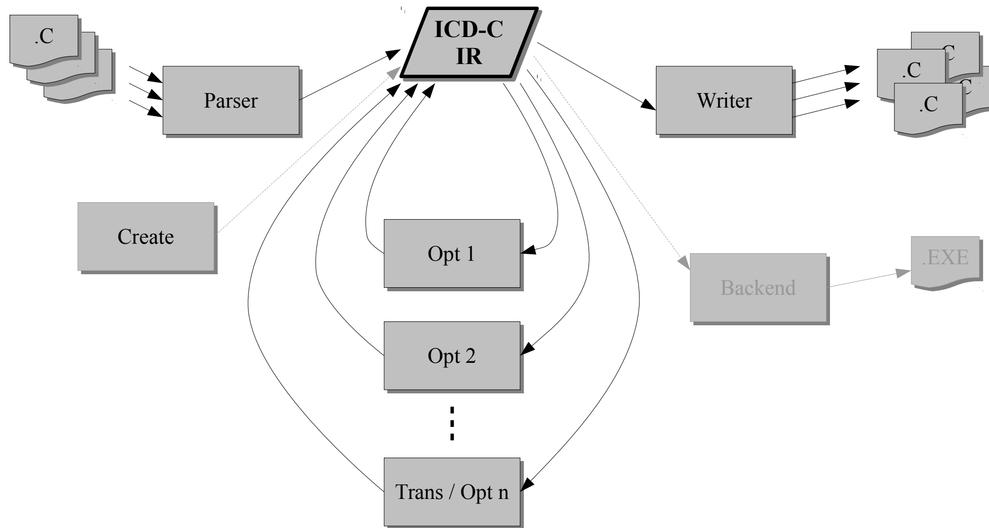


Figure 2.4: ICD-C workflow.

Figure 2.4 shows commonly-occurring ICD-C workflow scenarios. A typical method to construct ICD-C intermediate representations is to parse a textual C program representation. A corresponding parser is provided within ICD-C. Besides the parsing-based method, fully API-driven construction of intermediate representations is supported. Basically, the same API, as used for modifications of an IR, can be used to construct such an intermediate representation from scratch.

The intermediate representation itself and surrounding services are implemented in the C++ language. Therefore, the IR and corresponding API are exposed to the user in an object-oriented manner. Consequently, modifications performed on the application code via ICD-C are performed as modifications on the IR object graph or as modifications to properties of particular objects in this graph. Typical modifications are replacements of expressions or statements. Also more complex transformations such as inlining or exlining of code fragments can be performed in a semantically correct way with limited effort.

Once the application code has been modified accordingly, the optimization technique may decide to write-out the intermediate representation as a plain-text C program. ICD-C provides all the necessary methods to dump each compilation unit into a text file which can be accepted as C program input by a subsequent optimization or compilation step. If applying multiple optimization steps in the context of this framework, such a conversion between textual representation and the intermediate representation can be avoided on intermediate steps. Since the ICD-C framework is based on the common-object model of MACCv2, the serialization and persistent data retention methods can be applied to store and later reconstruct a snapshot of an IR. The advantage of this data retention method is the seamless retention of all analysis and optimization-related annotations.

Therefore, this is especially beneficial in the case where application code is being passed between several optimization steps, which need along with the application code some additional input. In combination with the system-modeling approach, the MACCv2 framework uses this serialization-based storage method automatically. Since application code in the system description is always related to a particular processing unit, or a class of processing units, it is inherently part of a system description. Therefore, it is stored and reconstructed within corresponding system-model operations.

2.7 MNEMEE Project

According to the name "**M**emory ma**N**ag**EME**nt technology for adaptive and efficient design of **E**mbedded systems" the European Commission-funded MNEMEE project [33] targets development of optimization techniques which take the memory-subsystem properties into account. Since the work proposed in this thesis has been used as the integration foundation in MNEMEE, detailed presentation of this project is given next.

2.7.1 Project Overview

The MNEMEE project aims at the provision of automatic optimization techniques which increase system efficiency in terms of energy consumption and runtime, while maintaining unchanged input/output behavior of such a system. Especially, tight development time frames, reuse of previous designs and adherence to standards limit significantly the set of options a designer may chose from once implementing new appliances. On the other hand, the computational complexity increases along each generation. These contrary demands force a prolonged optimization and tuning phase which tries to fit the software in an efficient way on the hardware platform. Mainly due to the complexity of such embedded software, performing such an optimization manually is extremely tedious and error prone or even impossible task in a given time/budget frame. Therefore, MNEMEE assists developers in this optimization phase, by providing a source-to-source optimization framework which offers a set of state-of-the-art optimization techniques which address efficient dynamic allocation of application data items as well as mapping static data items to appropriate memories. Since current system designs often expose multiple processing units, a further topic addresses automatic parallelization and mapping of function blocks of given source code to these processing units. These optimization techniques take runtime, memory footprint and energy consumption into account. Depending on the actual optimization a trade off between these dimensions is determined. Since these analysis results are exposed to the designer, MNEMEE supports design space exploration approaches.

Summarizing the project description according to its description of work: "The MNEMEE project will deliver all the necessary design methodologies, heuristics and prototype tools to enable the fast exploration of the huge dynamic and static design space."

The project outcome can be classified in following topics:

- Source-to-source transformations which aim at the reduction of the most important design metrics: Memory footprint and energy consumption.

- Perform these transformation and optimizations steps in a coordinated way, which follows a multi objective cost function.
- Since these optimizations do not have to be performed manually, MNEMEE reduces design time effort significantly and allows shorter time-to-market.
- MNEMEE analysis and transformation steps are implemented in a platform-independent way, therefore applicable to future platforms.

2.7.2 Toolflow Structure

According to the MNEMEE project, the target is an automatic toolflow, which applies optimization techniques in a predefined sequence. This section introduces the structure of this toolflow. Since the MACCv2 framework inherently supports such a development process it has been a reasonable choice to use it as the foundation for the MNEMEE toolflow. The actual benefits can be observed in more detail in Section 5.5.1.3.

Combining optimization techniques contributed by each project partner is a challenging task. Especially in the context of an European Commission funded project, such a geographically and organizationally distributed development process requires clear interface definitions and separation between each optimization approach. According to MACCv2, each optimization approach is a self-contained entity in this project. Each of these entities uses a common interchange format. Therefore, passing of required information from one optimization technique to another via methods provided by this framework is feasible. Despite the separation, each optimization technique is expected to have the same notion of the target platform. The MACCv2 framework provides extensive means of system modeling.

The actual ordering of processing-steps results from causal dependencies between these steps. One example is the tool (MPMH) which performs code transformations according to a parallelization specification. This specification is provided by the taskgraph extraction. Therefore, a natural ordering between these processing steps exists. Further dependencies have been analyzed. Dependencies according to the abstraction level on which an optimization step performs its task have been found. Optimizing dynamic data structures is less architecture-dependent than mapping tasks to processors. Similar relations exist between mapping of tasks, and per-task data assignment to available memories: Assignment may be performed only after a mapping exists. On top of these dependencies, the goal of achieving an energy-efficient execution of optimized application code exists. Therefore, it is of utmost importance to ensure that each processing step in the sequence does not disrupt or oppose gains achieved in previous steps.

According to these prerequisites, the integrated toolflow applies the processing steps in the order depicted in Figure 2.5. This structure honors tool dependencies due to one processing step providing meta-data to the next level, as well as enabling alternative execution paths, as shown in the case of the mapping step.

Each box in Figure 2.5 depicts a tool in terms of the underlying MACCv2 framework. They also correspond directly to processing steps described in the subsequent list.

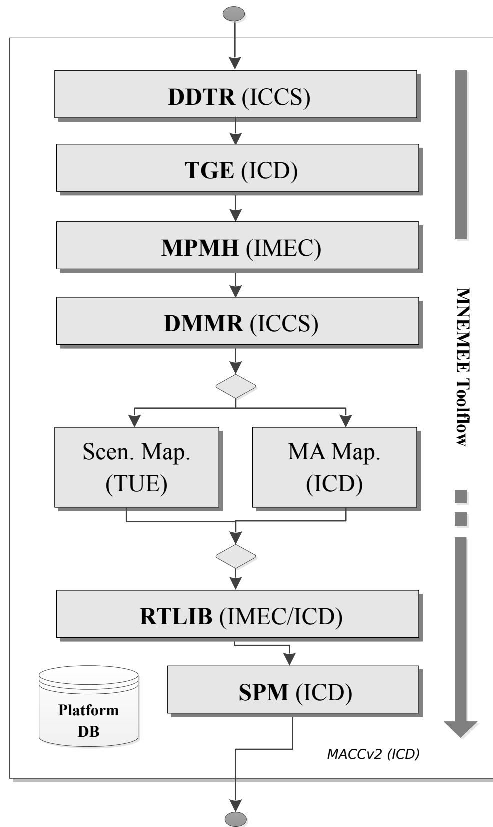


Figure 2.5: MNEMEE toolflow overview [33]

Along the toolflow work path, a short overview of the transformations and optimizations performed in each processing step is presented next:

- Processing begins with the optimization of dynamic data types (DDTR). The transformation performed at this point adjusts the implementation of dynamic data types according to access patterns recognized for each particular data structure.
- The next step in the processing sequence is the task graph extraction (TGE). Up to this point sequential code is assumed to be processed in the toolflow. This step computes a cost efficient parallel representation of the code. The resulting code contains additional labels marking parallel parts plus additional meta-data describing the relation between these parts.
- Right after the TGE step, the MPMH tools are executed. The MPA part is responsible for actually implementing the parallelization found in the TGE step. MH performs initial transformations of static data (e.g., array splitting) to support a later mapping to scratchpad memories.

- Since static data has already been considered by MH in the previous step, dynamic data assignment also has to be taken into account. This will be done in the DMMR step. This step optimizes the allocation strategy for dynamically requested heap memory.

- The mapping step assigns the code generated in previous TGE and MPA steps to particular processing cores. The MNEMEE toolflow offers two alternatives at this point. On the one side, there is a scenario-aware mapping tool available and on the other side there is a memory-aware mapping tool available. The scenario-based mapping tool considers the dynamic behavior of an application when allocating resources. The second mapping approach performs an allocation of tasks to processing units under consideration of memory usage and communication overhead.

- Similar to the TGE/MPA pair, the mapping step and the subsequent RTLIB step cooperate. Whichever mapping decision is taken in the mapping step, it has to be realized later on in the code by the RTLIB step. The name of this step is related to the hardware abstraction layer user in the MNEMEE toolflow.

- The final step in the MNEMEE toolflow performs the actual allocation of static data objects to memories. Previous steps make assumptions on the location of memory objects, but do not actually allocate these objects to particular memories. It is optional for them to provide their assumptions as hints, but the final decision is taken in this step.

Between every processing step in the toolflow, the application code can be extracted from the intermediate system representation. Doing this after the allocation step would result in the representation of the final, fully optimized, code.

2.7.3 Project Outcome

The MNEMEE Project has developed an optimization toolchain which executes a sequence of source-code transformations targeting the goal of energy-consumption reduction and/or runtime reduction. According to the toolflow structure, each processing step targets a specific type of optimization technique. Since interdependencies between processing steps exist, the order of applied optimization and transformation steps is driven by code structure requirements, presence of annotations and target-platform mapping types. The sequence chosen for the MNEMEE toolflow follows these dependencies. Data structure optimization is typically possible quite early in the processing sequence. No particular mapping or parallelization is mandatory to find a suitable data type refinement. Nevertheless, subsequent steps could expose causal dependencies. Transformation steps which implement optimization decisions have to be subsequent to corresponding decision-taking steps, in general following top-level order is required: First, a parallel model of the initially sequential application code is needed. Completing such a model implies addition of communication and synchronization methods into application code. Later, this parallel representation of an initially sequential code is used to find a valid

mapping of each task to a processing units. Finally, a reasonable usage of tightly coupled memories is only possible once such a mapping has been chosen. Only at this level is the necessary knowledge of available memory-allocation options for each particular data items available. Therefore, applying processing steps in the order proposed for the integrated toolflow turned out to be a reasonable approach.

According to the MNEMEE project report D5.3 [58], describing the application of this toolflow to a combined benchmark consisting of a MPEG4 encoder with a subsequent network transmission layer, the overall toolflow achieves remarkable energy and runtime savings. Refer to the D5.3 report for individual processing-step results. For the purposes of this thesis, the concluded results stated in this report are cited next:

> To summarize the benefits observed by the MNEMEE toolflow for the benchmark applications, we can claim the following:
>
> **Design Time:** The design time for the parallelization and data optimization of the benchmark application have been reduced by 80% by assuming the involvement of a skilled application developer.
>
> **Memory Bandwidth:** The memory bandwidth for the application can be reduced by 54% for the dynamic part of the code. However, such a decrease is accompanied by an increase in the memory footprint.
>
> **Memory Footprint:** The memory footprint reduction for the application is really negligible (less than 1%). Such a low benefit can be attributed to the fact that this particular application was designed by a group of skilled application developers who already made sure the memory footprint of the data structures are low.
>
> **Execution Time:** The execution time for the application time can be reduced by 30%.
>
> **Energy Reduction:** The integrated toolflow optimization reduced the energy consumption by 30% using an analytical estimation.

According to the individual results presented in D5.3, applying individual tools results in diverging optimization results. Since each optimization step has different optimization goals, the individual results give higher gains in one optimization direction while resulting in less effective or even in performance decrease in another dimension. A typical example is the parallelization step. Moving from sequential code to a parallel implementation which utilizes all processing cores available, typically decreases the required runtime drastically but at the cost of increased energy consumption. Nevertheless, the integrated toolflow shows the benefit of applying all the processing steps in a sequence. Even though the parallelization step introduces increased energy consumption, subsequent memory-allocation steps outweigh this disadvantage and give a combined result which is optimized in all directions; runtime, energy consumption and memory footprint.

The fully automated integrated MNEMEE toolflow has shown a further advantage once a broader set of application codes is processed. Iteratively exploring application-code variations is possible without user interaction, effectively reducing design time effort

significantly. Further, even though optimization benefits of individual optimization steps do not accumulate, the MNEMEE toolflow shows that applying several optimization steps in a sequence improves coverage of the optimization space. Typical application code shows a preferred optimization direction. Either the general application domain (i.e. multimedia code) or the coding style opens up optimization possibilities for a particular optimization technique while eventually rendering other optimization techniques pointless. Therefore, such an integrated flow can adapt to the application type and cover several application domains.

The subsequent citation taken from the MNEMEE project report D5.3 summarizes well the successful integration:

> To summarize the evaluation of the MNEMEE toolflow and optimizations developed during the whole project, it can be concluded that the MNEMEE has been successful in integrating a number of independently-developed tools by different partners in an efficient and intelligent way. The overall optimizations in terms of design time, memory bandwidth are in line with the claimed benefits at the start of the project. For memory footprint or energy reduction, the benefits are not as high as expected at the start of the project. However, the discrepancy can be explained from the constraints from the application used in the experiments as well as the platform simulator used. This is not a bottleneck for the MNEMEE toolflow itself rather an implementation issue. As described in Deliverable 6.4 and 6.5, the implemented toolflow has a well-defined path to be converted into a useful design tool for the future embedded developers.

3 Design Specification

3.1 Introduction

Memory-aware source-level optimization techniques alter memory-access patterns of an application under optimization to achieve lower runtime or reduce energy consumption. In general, such optimization techniques need some cost model to decide on the transformations to perform. Fundamental architecture-dependent data guiding these cost models is provided by the system-modeling approach proposed in this thesis. Optimization techniques exploiting such architectural properties benefit from the presence of a system model which is capable of delivering the required properties right on time. In particular, this implies the accessibility of such system properties while the optimizations are being executed. In line with this requirement, the MACCv2 approach provides a database-like access to the target-system description. A key component here is the so-called access query, which encapsulates the request imposed by an optimization technique and links it to the computed cost values.

The novel system-modeling approach proposed here features an architecture description approach which can be classified as a structural PMS model according to Bell and Newell [6]. In contrast to the primary focus on description of processing cores and their instruction sets in common architecture description languages (ADLs), this one features a fullfledged system model including details on the memory subsystem. At the core, this is a structural system and memory-hierarchy model enriched with semantical information.

As various optimization techniques may require various precisions of target-architecture descriptions, this approach neither requires a fixed set of component types nor a fixed set of properties per component. Therefore, in contrast to previous ADLs where a detailed model has to be developed in advance, in this approach only high-level structural information has to be provided for a new architecture. Once tools require more detailed representation, these details can be added to the model without losing backward compatibility.

Developing source-level optimization and transformation techniques results in a significant amount of recurring work. This thesis proposes the MACCv2 framework which aims at reducing this recurring overhead by provision of a common foundation for a wide range of optimization and transformation techniques in the domain of memory-aware optimizations. The practical implementation of the system-modeling approach proposed in this thesis is tightly integrated into this framework. Several further services are provided within this framework as well. The complexity of these services starts at a very basic level up to complex tasks, like processing-step encapsulation.

Reducing the implementation overhead for a particular optimization technique is a prevalent requirement, obviously the minimal implementation effort can be achieved for a particular optimization technique if every previously implemented processing step

can be seamlessly reused. In general, implementing optimization techniques based on a common framework is the preferred approach to increase the chances for successful reuse. The MACCv2 framework presented in this thesis takes such an approach a step further. Providing means of processing-step encapsulation and data exchange facilities helps in designing highly reusable self-contained optimization and transformation techniques.

The memory-hierarchy description presented here is defined as an object-oriented C++ API usable within an optimization or analysis technique implementation. This fits well with the common approach to develop such techniques in the C++ language. Especially, within the context of the optimization technique development framework, where the application code is represented as a set of C++ objects constructing an abstract syntax tree, the C++-based API for this system description is a natural choice.

This chapter initially motivates development of a system-modeling approach as part of an optimization framework focused on memory-aware source-level optimizations. First, a closer look at the system-modeling approach is taken. This includes description of structural properties as well as in-depth presentation of the database-like access to target-platform properties.

In the following sections the set of services provided within the framework is presented to the reader. Especially, the processing-step encapsulation and toolchain construction methods are at focus. To cope with the complexity of current optimization techniques, developers tend to implement stepwise approaches which subdivide a task into a better tractable sequence of steps. The framework presented in this chapter plays the coordinating and integrating role in such a scenario. Each self-contained processing step is defined as a tool in term of this framework. The tool interface captures the dependencies between such tools, enabling even fully automatic construction of toolchains.

Finally, a conclusion is drawn summarizing the properties of MACCv2 framework and the system-modeling approach.

3.2 Motivation

The system-modeling approach presented in this thesis has been developed in motivation for a versatile system model capable of provision of all necessary system properties to any processing steps in a source-level optimization toolchain. The goal is to support design of optimization techniques which are applicable to various target platforms without modification. To achieve this target-platform independence, besides a common system description, provision of several services to the optimization technique is required. Therefore, the system-modeling approach has been embedded into a framework. Later on in this chapter these framework services are presented as well.

Target-Platform Knowledge for Source-Level Optimizations

Observing the requirements of present source-level optimization techniques, a general preference for structural system descriptions can be concluded. In an initial step, a technique going to exploit particular system properties requires a coarse grain overview

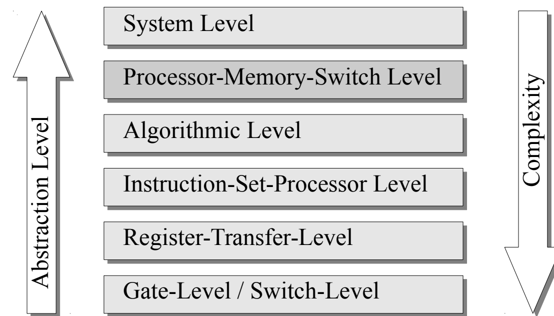


Figure 3.1: System-modeling abstraction levels.

of target-platform structure. In the case of a memory-access optimizing technique, this could be the set of present memories and their sizes. For a technique performing parallelization, the counts and types of processing elements would be of initial concern. In any case these properties are provided naturally in a structural model. In contrast to this, behavioral models put their focus on an executable representation of a target platform. Within this scenario it is tolerable to obscure the semantical and structural properties, since the common usage of behavioral models, the simulation environment, applies defined access pattern to the system model and the focus is directed towards a precisely computed result not to the knowledge of how this result has been achieved.

With respect to the abstraction level, there are two contrary demands imposed to the system-modeling approach. On the one side, a coarse-level system description is required to provide an initial overview of a target platform. On the other side, to achieve reasonable results, detailed knowledge of target platform is needed later on. As the basic requirement to provide a target-system overview has been previously identified, the initial abstraction level to consider would be the system level, as depicted in Figure 3.1. Unfortunately, describing only the components of a system would be appropriate only for a too limited set of optimization techniques. In many cases, and especially in the class of memory-access optimizing techniques, more detailed knowledge of the structure of a system is required. As a practical example within that optimization technique class, having only the knowledge of presence of a particular memory and its size may lead to invalid optimization decisions. This could be the case if the memory is shared in a MPSoC system between several processing units, each unit having access only to a dedicated fraction of that memory. To be able to provide reasonable data for such system configuration, properties of links between system components have to be considered. Therefore, compared to a plain system-level model, a slightly refined model is required. This type of system models is considered to be located at the PMS level, as defined by Bell and Newell [6]. Within that model, explicit focus on interconnection links (the "switches") is put. According to this classification, the system description proposed in this thesis also puts prominent emphasis on the interconnection description. Especially, modeling of address mapping rules between components and interconnection channels

provides the means of determining the address space layout as seen by each component in the system.

Besides minimizing effort for optimization technique developers, the system-modeling approach is required to minimize the effort also for the system-description developer to gain acceptance. From that point of view modularity is the primary goal. Describing components and links between these components in a self-contained way, which helps to avoid implicit influence on a particular component property due to its interconnection state, is a major motivation for this system-modeling approach. To motivate this modularity requirement, the most common disadvantage of state-of-the-art system descriptions is sketched next. In common approaches, the system description incorporates a concise list of system component properties. Within that list values are stored in a processing-element-centric way. As an example, memory-access latencies will be represented as accumulated values including bus and processor overheads. Such representation may be acceptable as long as the values can be easily computed for a given platform and this target-platform structure does not change often, otherwise recomputing these values may introduce significant overhead, eventually rendering these representations useless for design space exploration. The approach presented here avoids such issues by providing means of expression of component properties locally and defines rules how these properties can be combined to system-wide values. Providing a system-modeling approach with such compositional value representation fulfills both requirements imposed within that section. First, a system-model developer can perform development on a per-component basis. This enables construction of model component libraries from which components can be plugged together to form a MPSoC system description. This resembles well current preferred platform-based MPSoC development approaches.

A Solid Framework Foundation for Efficient Optimization Development

Even developing optimization and transformation techniques based on a well-defined system model is a challenging task. Especially, once it comes to implementing cutting-edge research ideas, the developer's focus is located at the core algorithms. Infrastructural work is often an additional burden, which prolongs the time until results can be generated and published. Even worse, developers tend to perform this infrastructural work with minimal effort, which limits the reusability of an optimization technique. Developing optimization techniques based on a framework approach mitigates these problems. Several positive effects aggregate enabling much reduced infrastructural overhead. In particular, an ad-hoc framework implementation would be performed in two steps: Identifying a set of APIs for common recurring tasks and moving corresponding implementation into the framework. The resulting long-term effect of a framework-based optimization technique development provides improved integration with other approaches developed based on top of the same framework. Secondly, the separation of optimization-technique-specific code and a common framework-based API library implies an interface line between them. Along this interface, parts of code can be combined or the underlying implementation of these API methods refined, distributing the improvements to all optimization techniques. Even though this iterative approach provides some benefit, a more structured

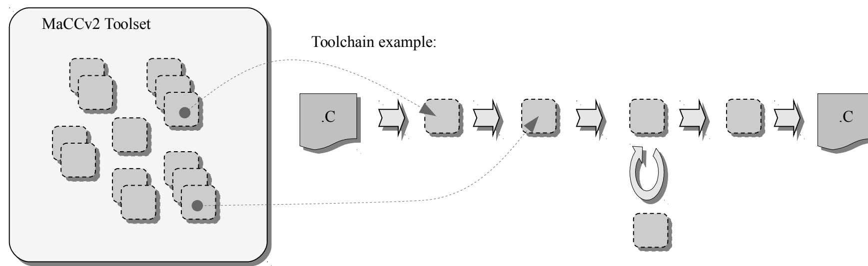


Figure 3.2: Typical toolflow construction.

framework design improves the optimization technique development process even more. The MACCv2 framework proposed within this thesis targets such a focused development of a source-level optimization technique infrastructure. Both previously identified targets, integration and reuse, are at the focus of this framework.

Observing recent and past source-level optimization techniques, in most cases a stepwise approach can be concluded. This may be quite subtle and simple as the division into an analysis phase and a transformation phase, but it may also be complex as in the case of the MNEMEE Project [33] where several types of memory-access optimizations were performed in a sequence. In such a toolflow scenario the overall outcome may be seriously affected by inconsistent system models between processing steps.

A set of services has been identified which are necessary and beneficial for source-level optimization and transformation technique development. As indicated right in the preceding section, a common notion of the target platform for which optimizations are performed, and a common application-code representation are of utmost importance. Both services are provided by the MACCv2 framework. The common target-platform representation is provided according to the system-modeling approach presented in this chapter. The common application-code representation is based on the ICD-C compiler development framework [3]. The application code is parsed and transformed to an abstract syntax tree. This representation is well-suitable for source-to-source approaches, since the application-code semantic and syntax is preserved at a very high level. Essentially, this enables reconstruction of application source code from such an abstract syntax tree.

Providing a common notion of target platform and application-code representation is accompanied by further services. The first one is a very fundamental service provided by the framework. It implements methods for persistent data retention and communication of results between processing steps. A common service for data storage saves much recurring work, which would be invested into implementation of file-based configuration methods and input file parsers. Further service, especially in the context of integration support, is an optimization technique invocation interface. In terms of this framework each optimization and transformation technique is named a tool. Such tools can be called in a predefined way. Encapsulating each optimization into such a uniform tool representation is very useful once forming toolchains of optimization techniques becomes

necessary. Automatic invocation and seamlessly exchangeable user interfaces are only two benefits of this approach. An exemplary application scenario for this feature is shown in Figure 3.2. The blocks in this figure represent self-contained transformation, analysis and optimization techniques which have been combined to form a source-level optimizer. A practical example is the MNEMEE toolflow presented in Section 2.7.

The work presented in this thesis is expected to support rapid source-level optimization and transformation technique development. Especially, focusing on a common notion of target-platform properties, integration and reuse. To achieve this, a wide range of services has been bundled into a framework, starting from platform and application-code representation up to user-interface issues. The optimization developer does not have to deal with infrastructural work, but can focus on the actual optimization technique implementation.

3.3 System Description

The system-modeling approach proposed in this thesis exposes a set of properties beneficial for development of source-level optimization techniques. From the top view perspective it resembles a structural system-level architecture description language. Within that description, the target platform is represented as a set of components and channels. Components expose ports which link them to correlated communication channels. Having a closer look at these links, emphasis is put on address space relations and corresponding mappings. With that level of detail the overall approach can be classified as a PMS-level architecture description. In more detail, the system description focus on these properties:

Expose a native language interface: In contrast to common plain-text-based description languages, this one exploits features of the C++ programming language to represent the system model. Both, from the perspective of a system-model designer, as well as from the perspective of a source-level optimization technique developer, the system-model description is accessible as a C++ object graph. In combination with the persistent storage method of the MACCv2 framework this joins the advantages of both approaches. A particular system model can be stored to disk in a self-contained file while the flexibility of a programming language-based interface is still maintained. Especially, C++-based inheritance provides a natural way to classify the set of components and channels. Furthermore, in combination with the ICD-C-based source-code representation, which exposes the application code as an abstract syntax tree consisting of C++ objects, a fast and straightforward way to express relations between system-model objects and source-code object exists.

Define an abstract system model: System-model components are in general expected to relate to physical components of the target platform being modeled.

As this system-modeling approach is located at an abstract level, the basic component is regarded in first place as a black box. Therefore, the model does not enforce provision of any information on the internal structure of a component. The only requirement is to provide ports on the surface of it to give handles for linking them via communication channels to other components. Subclassing is used to represent various types of components of the target platform. There are component classes for processing units, which further specialize into particular processor models. Similarly, there is a class of components for any type of memory. An example for further specialization in that class of components are scratchpad memories. The subclassing approach opens up the possibility of having varying sets of properties per component. For example, processing-unit components may have some application code associated with them which indicates this code is going to be executed on that particular processing unit.

Keep component interaction at focus: Channels are the second fundamental building blocks of a system description according to this model. They provide links between components. Due to the abstract nature of this modeling approach, in general channel objects are opaque in this model as well. Similar to components, channels may also define address spaces. Address spaces will provide a distinction between possible transactions performed via a particular channel. In contrast to components, channels do not expose ports. The channel object itself is the entity to bind communication paths to. Therefore, the basic model structure allows channels to link an unlimited number of components. Nevertheless, specialized channels for direct links and buses exist, denoting beside other properties the expected maximum number of connected components. Further specialization is applied to identify platform-specific communication links.

Provide a memory-subsystem-centric view: According to expected application scenarios for this system model, emphasis is put on the modeling of target-system's memory layout. In line with the requirement for modularity, the approach chosen here has to be compositional as well. To achieve this target, a structured address space modeling has been chosen. Each component and communication link has a set of address spaces it is aware of. An address space denotes an integral value range associated with an identifier. Furthermore, symbolic address spaces may provide additional symbolic identifiers for particular values. This is helpful for modeling register sets. A set of mapping rules, describing the translation between component and channel address spaces is a property of ports. The translation direction reflects the initiator-target scheme as occurring in the actual system being modeled. Since these local descriptions can be combined on request to build the well-known memory maps, there is no need to provide them separately by a system designer. This is superior to common system models used these days.

Provide system-wide properties out of locally-scoped definitions: The system-modeling approach aims not only at provision of structural system models, as described up to now, but also at offering comprehensive interfaces to system-wide

properties of modeled platform. To avoid the downside of such data being hard coded in some type of table, a computation method is proposed, which allows for deriving such properties from information available at component level. The system-wide properties available this way are denoted as an aspect. Common examples for such aspects are latencies or energy-consumption values. Although these example aspects are represented as numerical values, they are not limited to such in this system-modeling approach.

Query-based interface to these system-wide properties: Optimization and analysis techniques based on this system description use such computed properties to guide their decisions. With respect to mentioned latency values, the amount of cycles required to access a particular memory location will typically guide immediately some allocation decisions. As these values vary depending on the access trajectory in the system description, in general a request for an aspect value is related to some access pattern as initiated by a system component. There are numerous options how to request such aspect values. Size, access type, best/worst-case distinction or a complete access sequence may specify a request. This flexibility covers a wide range of requirements imposed in common analysis and optimization techniques.

Modular approach for calculation of these system-wide properties: From the perspective of a system-model designer, aspect-value computation requires components to provide handlers for each aspect. Such handlers express the contribution of a particular component in the context of a request to the overall aspect value. Since aspect handlers are expressed as C++ objects, they may perform arbitrary computation on the aspect value. For common cases, predefined aspect handlers are provided, including the case of a simple accumulating aspect handler as often used for latency computation. Furthermore, also implementations of more sophisticated handlers exist, like those used for energy-consumption computation where an energy model is used to steer the aspect-value computation for each component.

To summarize the set of properties of the system-modeling approach presented in this thesis, the whole approach classifies as a novel structural system-modeling approach located at an abstract PMS level. Focus is put on composability and reuse of components and channels while maintaining a minimal effort for initial system-description implementation. Besides pure structural information, the system-modeling approach has been designed to support the user in determining system-wide properties without the need for an explicit definition of these properties. This renders the approach well-suited for optimization and analysis steps which require accurate information on whole target system, like memory-aware optimizations as well as design space exploration approaches which require fast and simple means of target-platform description modification.

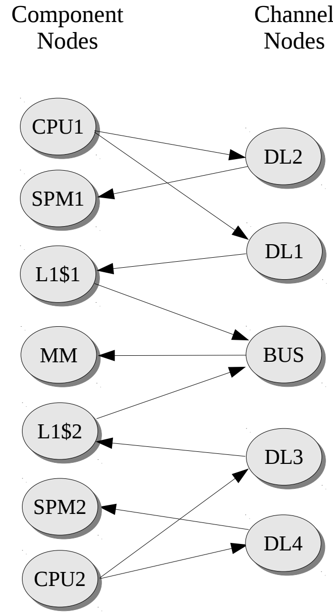


Figure 3.3: Graph structure example.

3.3.1 System-Model Structure

The structure of the system-modeling approach proposed in this thesis is introduced next. It starts with a graph representation, which captures precisely the relation between system-model items. Followed by a more visually appealing, component centric representation, which resembles well the typical way or target-platform visualization.

Graph Model

The system description presented in this thesis resembles a directed bipartite multigraph. The system-modeling approach imposes intentionally no requirement on the graphs being simple nor being connected. Nevertheless, common real-life MPSoC models will form a simple graph, with its isomorph undirected graph being connected. According to the graph definition, the set of nodes consists of two node subsets representing components ($c \in C$) and nodes representing communication channels ($l \in L$). According to the requirement for the graph being bipartite, the set of edges (E) may only connect nodes from different node subsets C and L, therefore, restricting every edge e to $e \in \{C \times L, L \times C\}$. The direction of edges represents the initiator-target relation as present in the system being modeled. Components being initiators of communication activities on a channel will be connected by edges from $C \times L$, while components being targets will be connected to the corresponding channel via an edge from $L \times C$.

The multigraph property is required to express system structures where components connect to a multilayer crossbar communication channel. In this case, focus is directed towards the possibility to represent several parallel communication activities being performed concurrently. To be able to design system models with disjoint subsystems, the system-model graph may consist of disjoint subgraphs. Therefore, even the isomorphic undirected system-model graph will not be connected. Considering undirected graphs here is required since common components expose an asymmetric behavior with respect to the initiator target scheme. Therefore, the requirement for path existence between every two component nodes for a graph to be connected will be violated in almost any system model. Therefore, the undirected graph is of much higher value once it comes to identifying disjoint subsystems in the overall graph.

Figure 3.3 depicts as an example a graph representation showing the architecture according to Figure 3.4. This architecture consists of two cores with local memories and caches and a shared main memory. On the left-hand side, nodes of the subset C representing system components are located. On the right-hand side, nodes of the subset L related to the communication channels are depicted. Edges across these node sets represent port connections according to the system structure.

Constructing an initial system description is basically a two step approach. First, collecting the component and channel nodes, and later on linking them via edges together. If some target-platform component or channel is not present in any system-model library, there are several options how to proceed with system modeling. The selected option primarily depends on the intended use of the system description. In the simplest case a more generic component class can be used instead, resulting in less precise property representation or a complete lack of particular property values. This may be a viable way, if only high-level structural information is required to perform the intended optimization or transformation steps. The opposite approach is to describe the missing component or channel as a new unique class providing a complete set of properties. Obviously, this may be a tedious task. Therefore, this system-modeling approach promotes an incremental approach. The initial effort consists of defining a new, type-specific component or channel class derived from the most specific known one. Initially, the set of properties available for that more generic component class is reused. Once more specific details are available, a dedicated component or channel class already exists, enabling easy modification of that class. Furthermore, with respect to the optimization and transformation techniques, at any refinement level, the components and channels have a distinguishable type.

Structural Representation

To provide a more intuitive access to system descriptions modeled according to this approach, a graphical representation is proposed within this thesis. To introduce the key visual items an example system representation is shown in Figure 3.4. Rounded boxes represent nodes denoting components in the graph representation. At the edge components expose handles, which denote the set of ports exposed to the system model. Components may have some key properties printed within the rounded box (i.e. name or

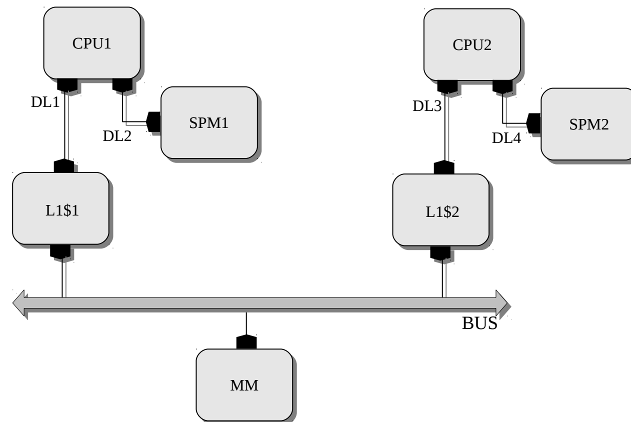


Figure 3.4: System-description instance.

class). The second set of nodes, the channels, have varying representations. In the case of buses or any other channel type capable of connecting more than two components, a broad bidirectional arrow is used to represent a communication channel. In the case of direct links no explicit representation is used. A line directly connecting ports of two components implies a direct-link node in the graph on the path between these component nodes. According to the system model, graph edges connect components to communication channels and vice versa. In the graphical representation also the relation to the port is explicitly shown. Each port connects to a single communication channel. Within the graphical representation a line is drawn between the port and the communication channel. To avoid cumbersome representations, channels do not have explicit connection points, but accept the connection lines on every edge. This matches well with the system model, where communication channels are also the connection targets by themselves.

With these building blocks a wide range of system models, including relevant state-of-the art MPSoC architectures, can be expressed. Starting from plain processor-bus-memory models via MPSoCs with hierarchical bus systems up to NoCs with their various topologies. Figure 3.4 shows a system model with bus communication channels at two levels. There are two local buses and a system-wide bus connecting both computing tiles to a shared memory. Vital to form hierarchical system models is the capability of components to be connected via different ports to several communication channels. In particular, such a component will have different roles on these connections. On the one side, the component acts as a target component for incoming accesses. On the other side, it will initiate subsequent accesses down the hierarchy path. As described in Section 4.3.1 presenting practical component models, this is exploited in component models for caches or bridging logic between buses.

The following subsections take a closer look at components, channels and further system-description items.

3.3.1.1 Components

Components are the model representation of self-contained entities in the target system. In the graph-based interpretation they are one subset of nodes in the bipartite graph. In general the system model is expected to be located at the PMS modeling level. Therefore, the mapping to entities of the target platform has to be performed at an adequate hierarchical level. Usually, this would be processing units, memories, caches or self-contained special purpose hardware blocks. A natural correlation exists to state of the art MPSoC designs, where a platform-based approach is commonly used. The component model of this system-description approach matches well with the platform building blocks used to form a MPSoC.

According to the needs of optimization or transformation techniques the system model is not limited to match exactly these platform entities. It may be useful to introduce a higher level of detail. An example would be an explicitly represented register file of a processing unit. In combination with a register set address space this would result in a system model suitable for register set usage optimizations. Another feasible deviation from a one-to-one mapping of entities of the target MPSoC to system-model components would be explicit modeling of large background memories (i.e. hard disk swap spaces). This could simplify application of memory-allocation strategies also to the main memory. Such allocation strategies usually assume an infinite fall-back storage location for all items which were out of scope or did not fit into the target memory. Such role could be given to this additional storage, pulling the main memory in the scope of memory-allocation and optimization techniques.

Component Inheritance

The set of components is structured as an inheritance tree. The root represents a generic class common to all components. The first level of subclassing distinguishes between processing units, memories, caches and further special purpose hardware blocks. In the memory domain further specialization identifies scratchpad memories. In general the leaves of this inheritance tree are component classes representing a particular model-specific device. In the case of processing units this could be a specific ARM core. Figure 3.5 depicts the component inheritance tree. For simplicity the figure shows only the components as available for a MPARM [59] platform description. Other platform descriptions exist, which would result in a different set of leaf component classes.

As part of the MACCv2 Framework this system-modeling approach is implemented based on a C++ API. Therefore, the component classification is performed by means of C++-object-class derivation. Each component type has a C++-object-class-based representation. The inheritance tree is preserved in the C++ API the same way. Due to the requirement for the inheritance relation to form a tree structure, no multiple inheritance is allowed as it would be in general possible within the C++ language.

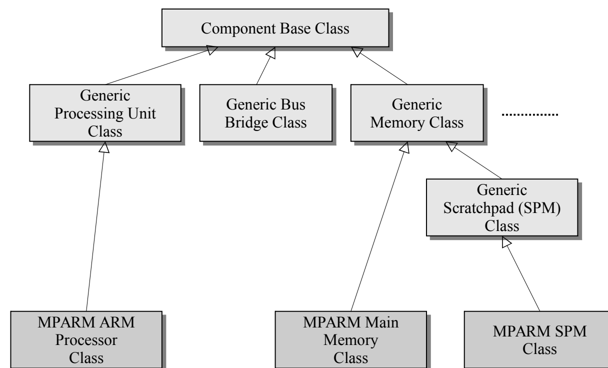


Figure 3.5: Component inheritance tree example.

Component Interconnection

Components are equipped with ports enabling them to connect to communication channels to form the structural system model. A generic component may have an arbitrary number of ports. Usually, the system-model designer defines the set of ports present in a specialized component. Each port connects to exactly one communication channel. At the abstract PMS level this would be sufficient to form a structural system model. To support the system-wide property computation, ports need to be equipped with some additional information. A port may incorporate a set of translation rules, which describes the mapping between locations in the address space of related component to locations in the channel's address space and vice versa. According to the design goals of this system model, there are some predefined mapping rules for the common case, but any arbitrary mapping may be implemented as additional rules.

With respect to the API, ports are also represented as C++ objects related to the corresponding component. The MACCv2 framework provides a container API to build these relations. The reader may refer to Section 4.2.1.9 for more details.

Component Property Model

The component structure described up to this point would give sufficient information to build a structural system model. The subclassing of component types enables the optimization technique to unambiguously identify the type of each component. Nevertheless, any property of such a component would need to be deduced from some external source. Even worse, optimization technique developers could be tempted to hard-code some properties into their optimization technique, rendering the whole approach bound to one particular architecture type. Therefore, major focus has been put in this system-modeling approach and underlying framework, to provide simple to use and flexible means of annotation of properties to any entity of the system description, including components.

Properties in context of this system model are any numerical values, enumeration of such, text strings or any object-oriented complex data structure represented as persistent object in terms of the underlying framework. Especially, the last one opens up a wide range of value representations including so-called smart data objects, where value computation will be performed at request time. Common to all properties is the need for an identification handle. Within this approach each property has a name. The name is unique with respect to the object (i.e. component) the property belongs to. For the set of practical property values please refer to Section 4.3.1 describing present component implementations.

With respect to the component model used in this system description, properties may either be related to a particular component instance, or to a class of components. In both cases the complete set of properties as described in previous paragraph may be used.

According to this system-modeling approach, properties defined for a particular component have to be local to that component. This requirement is vital for high reuse of components in various system descriptions. Nevertheless, there are properties which result from the overall system structure and are affected by several components and the communication links in between. One common example is the set of memory-access latency values as observed by a particular processing unit. To be able to represent such properties, the system-modeling approach introduces the concept of aspect handlers. Briefly described, an aspect handler computes the contribution of a component to the overall value of that property. For more details on the aspect handling refer to Section 3.4. From the point of view of a component, aspect handlers are self-contained entities which are related to that component. For each aspect-value type a single aspect handler per component may be defined. A good practice approach is to define component characteristics as property values and provide an aspect handler which refers to these values once associated with a component. This approach enables not only the reuse of components, but also simplifies design of new ones, since in addition a library of generic aspect handlers can be formed.

Tool Specialization

To simplify operation on system models and application code, the MACCv2 framework provides an abstraction layer which encapsulates such processing steps into so-called "Tools". Optimization techniques may apply such tools to perform common tasks. Nevertheless, the need for performing such tasks in different ways for each component in the system model may exist. A common example is a profiling step which collects memory-access statistics. In this case, different processing-unit types will require different simulation setups. Encoding profiling tool setups into the analysis technique implementation reduces significantly the reuse factor of such techniques. Therefore, the component model used in this approach provides a method to specify on a per-component basis a more specialized tool than the one requested by the analysis technique. From the perspective of a system-model designer, the effort required to exploit this feature is to provide the mapping between a class of tools (i.e. profiler) and the component-specific types.

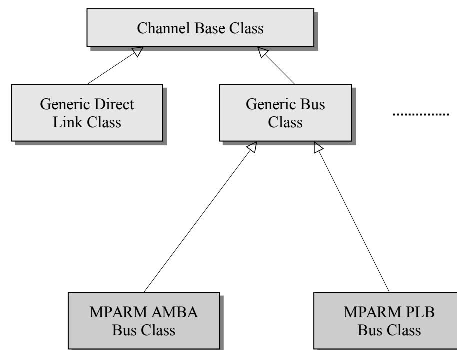


Figure 3.6: Channel inheritance tree example.

3.3.1.2 Channels

Channels represent the communication facilities present in the target system. This structural focus is different from the one present in behavioral modeling approaches, like SpecC [60], where channels represent an abstraction to the data transfers occurring within a target application. Besides components, channels are the second key building blocks of the system description presented in this thesis. According to the graph-based interpretation, they are the second subset of nodes present in the bipartite graph. In contrast to components, channels are not well topologically identifiable in the actual target MPSoC. According to the connection scope, they may form system-wide structures spreading across the whole MPSoC. Nevertheless, in a structural model, there is usually a set of related signals plus optionally some arbitration and access-control logic forming the communication channel. This part of the MPSoC is targeted in this system description as the channel model.

Channel Inheritance

As this system description is located at the abstract PMS level, it is not the primary goal of the channel model to capture the internal details of a particular communication channel. Similar to the component model, the primary goal for channels targets modeling of structural properties of the target system and provision of handles to attach properties to. Within that goal an identifiable communication channel type is of high importance. To achieve this, a classification and inheritance scheme similar to classification of components is used to identify communication channels. For the set of channels an inheritance tree can be constructed. Figure 3.6 shows such an example. Initial classification of channels divides the set of channels according to the number of allowed connections. The first class of direct links connects exactly two components. The second class denotes channels which are capable of connecting to multiple components. This kind of communication facility is in general a bus-based communication. Therefore, the common super class for this set of channels is denoted as a bus class.

In line with the underlying framework the API realization of this channel model consists of C++ classes and objects. To express the inheritance relations between channel classes a native C++ class hierarchy is used. The MACCv2 framework provides, in addition to the C++ language-based object representation, a rich set of reflection methods which support analysis of class hierarchy at runtime. The reader may refer to Section 4.2.1 for more details.

Channel Interfacing

In contrast to the component model where dedicated ports provide the connection points, channels are the connection targets by themselves. The decision to omit explicit connection points is motivated through the observed structural properties in real-life MPSoCs. According to the intuitive relation, identifying channels as a group of signals, there is no dedicated handover entity, instead the port is connected immediately to the channel signal. The system model assigns any interface logic and the related overhead to the port and therefore assumes this to be part of the component. This way, a clear distinction exists between which contribution to system properties are component-related and which are strictly communication-channel-related. Observing the common case of bridging hierarchical communication channels shows the advantage of this approach. Within this system model it is not possible to directly link two communication channels. Instead, a dedicated bridging component has to be introduced, urging the system-model designed to explicitly quantify the bridging effects and overheads, resulting in an improved overall system model.

Channel Property Model

Since the system-modeling approach does not target structural information only, but aims at provision of local and system-wide property values, property values have to be annotated to components and communication channels. The methods and the same flexible APIs present at components are available for channels. Section 4.3.1 shows the practical implementation and gives some example property values and their annotation methods.

Excessive reuse of system-model components and channels is of high importance to this system-modeling approach. Therefore, the restriction has been imposed to provide only property values which are local to a particular component. The same applies to properties attached to communication channels. This simplifies introduction of such channels into new system models, as its properties do not have to be adjusted to new application scenarios. Nevertheless, as observed for components, the need for system-wide properties still exists. To be able to represent such properties the system-modeling approach introduces the concept of aspect handlers. Both, components as well as channels, may provide a contribution to such values. Therefore, both have the same APIs and concepts for relating aspect handlers to them. For more details on the aspect handling refer to Section 3.4.

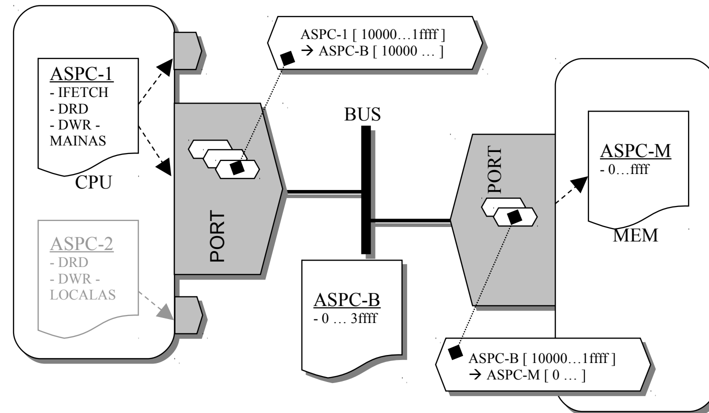


Figure 3.7: Address spaces occurrence example.

3.3.2 Address Spaces

Starting at the most generic definition: Address spaces denote a finite address range associated with an identifier. As widely used within the MACCv2 framework, identifiers are alphanumerical names. Addresses within this model are either integer numbers or an ordered set of symbolic values. Between address spaces and addresses a mutual dependence exists. Addresses always relate their values to an address space, this provides the context for that address value. In the opposite direction, an address space defines its range in terms of minimal and maximal addresses. The purpose of address spaces is to provide a uniform way to denote storage locations in the system model. Address spaces get their semantical value, once they are associated with components or channels. Each component or communication channel may have several address spaces associated with it. Depending on the assignment, address spaces serve different purposes. For components which expose initiator ports, they express a location as being actively accessed by some control logic within that component (i.e. application program in the case of a processing-unit component). In the case of components exposing target ports, an address space denotes a target location within that component (i.e. in the case of a memory component a particular storage location). For channels, address spaces are used to distinguish between various access types on that channel (i.e. some systems distinguish between I/O and memory accesses being transported on the same bus. Typically such a bus would have two address spaces.)

For address spaces, more precisely the set of addresses denoted by them, a wide range of operations can be performed. These operations include arithmetic operations and comparisons up to set-based operations which perform actions like intersection or joining of two sets. To be able to represent huge sets of addresses, the often observed locality is exploited to group addresses into ranges and lists of such, enabling a compact representation of address sets of which explicit enumeration would be intractable.

Address spaces and their value representation also have, besides the advantage of a common storage location representation throughout various optimization techniques, a

key value for the computation of system-wide properties within this modeling approach. The mapping rules as described in the following subsection operate on address spaces of adjacent components and channels. Figure 3.7 depicts this situation as well as various possible address space occurrences.

3.3.3 Access Model

Once addresses within an address space have been defined, some notion of action performed on these addresses is required. As addresses denote a storage location in a target component, actions on these locations are typically reading, writing of data or fetching of instructions in an instruction-driven stored program machine. Usually, accessing data is performed in type-dependent chunks. Most likely these chunks are not of the same size as a single storage location denoted by an address. Therefore, besides the initial location, an access has to denote how many consecutive location are going to be accessed while performing the desired operation. Summarizing these requirements, a simple access in terms of this system-modeling approach consists of:

- A set of initial locations in terms of addresses.
- An alignment within that set of locations, to determine any possible initial location within a given range.
- The size of an access in number of consecutive address locations.
- The type denoting the action being performed. Currently supported actions are data read, data write or instruction fetch.

The address indicates indirectly via the address space the component or channel in which context this access is located. In case of an active component equipped with initiator ports (i.e. processing cores or DMA units) accesses within this model can be intuitively described as accesses being performed by that component. In case of target components, an access describes an action as observed by such a component.

The simple access as described here, is considered to be an atomic operation. Especially, regardless of the size, the action is performed for a given initial location on all locations within specified range simultaneously. This definition has been chosen, to provide an architecture-independent access model. Nevertheless, for large size accesses this usually contradicts the behavior of real hardware devices, where only a fixed maximum size access can be performed simultaneously. Therefore, a second type of accesses is introduced in this model. The sequential accesses consist of a list of simple accesses. The list defines the sequence in which these accesses are performed. Especially, in the process of access translation in a path enumeration as described in next subsection, simple accesses may be translated to an access sequence at next level. An example describes an initial access definition at a processing unit, which defines a read access to a 16 bytes array. According to the platform description the processing unit is connected to a 32 bit wide bus. Therefore, performing a 16 byte read via such a bus as an atomic operation

is not feasible. As a consequence, the mapping will convert this access to a sequential access with 4 items related to the address space of the bus.

Besides this automatic construction of access sequences, the opportunity to define access sequences already at the initiator level exists. Especially, since such an access sequence does not have to target consecutive locations in the address space, but may target arbitrary locations, this can be used to get more precise system-wide property values, if such sequences are known to the optimization technique. A common example of components benefiting from such a sequence of accesses are components which expose different behavior according to the recent access history (i.e. Caches or DRAMs). Here providing an access sequence while querying properties like latencies will give the opportunity to that component to compute its contribution more precisely within the scope of such a sequence. Please refer to Section 3.4 for more details on aspect handlers and system-wide properties.

3.3.4 Access Routing and Mapping

The structural system model consisting of components and channels and connections between them gives an overview of the target system. Even though components and channels are required to provide only local properties, various system-wide properties are at least of the same importance. The structural model can be explored to retrieve these values. Simple methods like iterating over the components or channels are a widely used approach. More complex methods take the actual connection paths into account. To support such recurring tasks, the system model incorporates mapping tables attached to component ports. Within these tables translation rules between address spaces are defined. Using these tables, paths can be enumerated in a system-model-independent way. On these paths various operations can be performed. Examples of such operations are invocations of aspect handlers attached to components and channels on such a path to compute system-wide property values. Refer to Section 3.4 for more details on the value computation.

The decision to relate path enumeration within the system model to address spaces is motivated by the commonly arising question within the context of application-code optimizations: Which benefit is obtained by taking a particular optimization decision? Therefore, initial questions for the costs in terms of latency, energy etc. have to be answered to compute such a benefit value. Usually, the granularity of such values is related to the application-code structure. Either data items, like variables or arrays of such or code items, like functions, are the items for which particular benefit has to be computed. Within that view, the relation to address spaces comes from the observation that size, location and the access sequence to such items are the input parameters to any kind of cost computation method. All of these parameters can be expressed in terms of addresses within a given address space. The address space is determined by the processing unit to which the application code is attached to and further depends on the capabilities of that processing unit. Eventually, the type of a particular application-code item has to be taken in account as well.

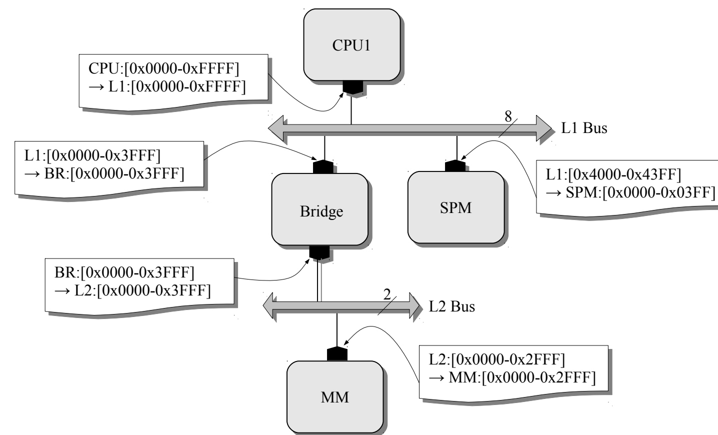


Figure 3.8: Access routing example architecture.

The required parameters match well with the definition of accesses according to Section 3.3.3. To abstract from the actual application code, these accesses are the basic items used for path enumeration. Either simple accesses or access sequences can be defined as an initial query. An iterative process is applied on these accesses until no more matching translation rules can be applied. Within that process, simple accesses may be split into sequences of accesses to satisfy the maximum access size limit of particular component or channel. Each translation step is recorded. In general, a full path enumeration results in a tree of accesses denoting in each child node the possible next hop on a path within the current system model. Keeping track of all possible paths is valuable once it comes to determine a single path within that tree according to some criteria like best/worst-case considerations.

For demonstration purposes an access is defined as follows (all addresses and address counts refer to the processing unit's address space. Each address denotes a 8 bit storage location):

- Initial address range: CPU:[0x1000-0x5FF0]
- Alignment: 1 location (each location)
- Size: 16 locations
- Type: Data read

Figure 3.8 show a simplified example architecture with a single processing unit and two memories at different hierarchy levels. Also the mapping rules are indicated in this example. According to them the processing unit can access both memories at different locations in its address space. Both memories are fully mapped into processor's address space. According to the hierarchy, both memories have interfaces of different bit width resulting in different maximum access sizes.

Requesting enumeration of access paths results in an access tree as depicted in Figure 3.9. Following the tree, several cases can be observed. First of all, the processing-unit component exposes only one initiator port with mapping rules for that address space. Therefore, only one successor node (the L1 bus) exists. Furthermore, the mapping is quite simple, as every address within the processor address space is translated 1-to-1 to an address within the bus address space. Nevertheless, one transformation is required to adjust to the maximum access size of the bus. Therefore, the access to 16 locations has been translated into a sequence of two accesses of 8 locations each. Once processing continues within the scope of the bus, there is no unique successor anymore. Actually, two components with one target port each are connected to the bus. The mapping rules cover disjoint subranges of the access' initial address range. Therefore, two successor nodes exist. As the lower address range up to 0x3fff is covered by mapping rules in the bridge component, the intersection with accesses in that range, results in an initial address in the range of 0x1000 - 0x3ff0 within the bridge address space.

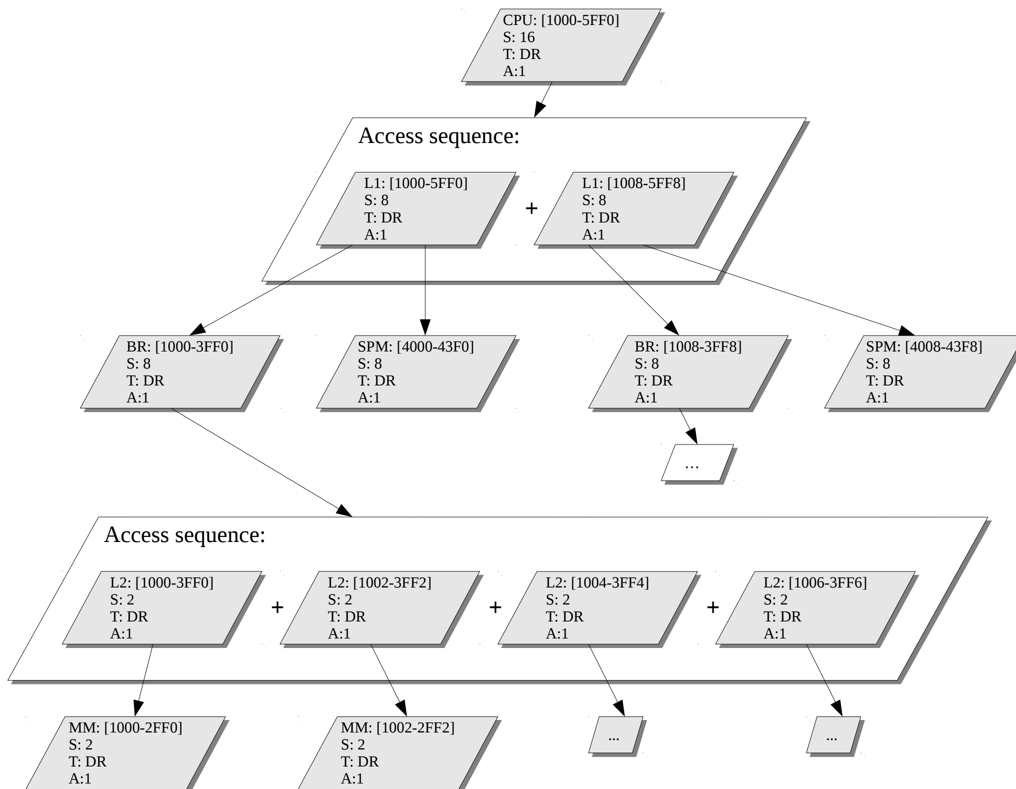


Figure 3.9: Access tree.

A second successor covers the range of the L1 memory 0x4000-0x43f0. Since the L1 memory does not have any initiator ports, this path terminates here. The bridge translates the access via its initiator port to the L2 bus. Here, a similar situation exists. A simple 1-to-1 mapping is performed, but another split into four accesses in total is

required. Finally, a single target port for the L2 memory with the range 0x1000-0x2ff0 is detected. Since there are no more unconsidered initiator ports, the tree is complete at this stage. In complex system models more effects may occur. The access type may be translated at each step. This often occurs if the processing unit distinguishes between data reads and instruction fetches, but the bus does not. Furthermore, in the presence of caches, a long sequence of accesses may follow down the path, when assuming a cache miss, and a preload of a cache line is required. Nevertheless, the basic concept of a tree holding all possible translations for a given request is still preserved. According to the general approach for this system model, to rely only on locally-defined information, the access routing follows the same strategy. Locally-defined mapping rules express the relations of adjacent components and channels. Furthermore, in most cases these rules are straightforward to implement. Nevertheless, the path enumeration enables a wide range of analyses at system level. One example is the computation of system-wide properties as described in the following section.

3.4 Aspect Modeling

Aspect modeling describes the method proposed for this system description to compute system-wide properties out of locally-defined data. The local definition is related to components or channels. According to the system model, each component and each communication channel may have various properties associated with it. These properties may describe numeric values, some tags or more complex data structures. Besides these local properties, corresponding values describing the properties of the overall system are typically of interest. Naturally, these global values depend on presence of components and the interconnections between them. To avoid inconsistent values and the need to update these values on each modification of the system model, the approach proposed here enables computation of such system-wide properties on request and therefore always provides up-to-date results.

Access-Based Computation

To compute such values, the access path enumeration and the corresponding access tree as presented in Section 3.3.4 are used. Based on these paths, the contribution of each relevant system-model component and each communication channel can be determined. To avoid the limitation to some single value-oriented contributions, like accumulation of values, each component may provide so-called aspect handlers, which can perform arbitrary operations. Therefore, the commonly used system-wide property representation as a numeric value is only one possible value type. Aspect handlers are designed to be self-contained objects which are attached to components or channels. This way the actual property computation is decoupled from the component or channel implementation. This allows for building-up libraries of aspect handlers for various property types which may be reused across several system models.

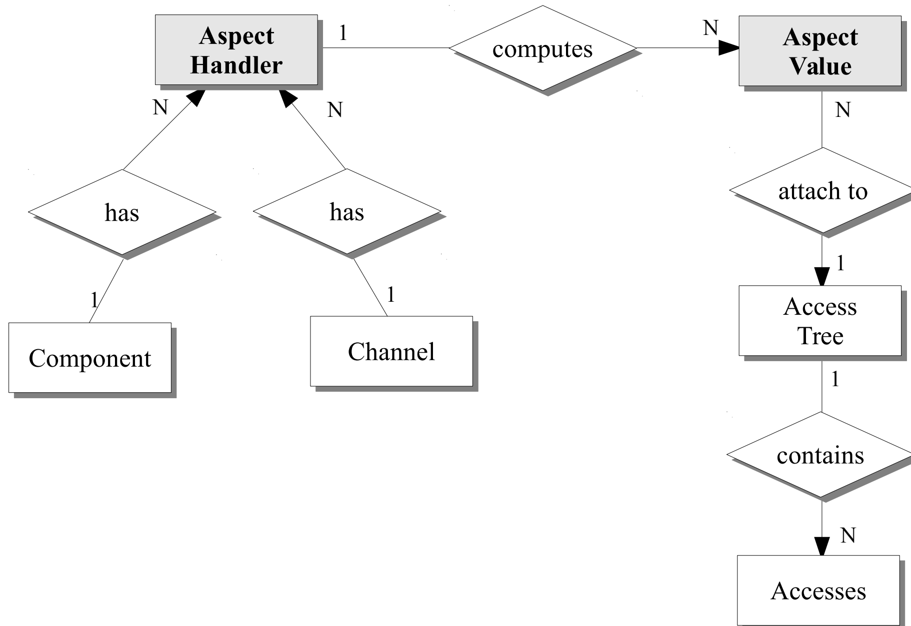


Figure 3.10: Aspect modeling overview.

Scenario-Aware Path Selection

Since in general the access path enumeration provides a tree structure, this may be problematic once a value related to one path is requested. Common examples are latency values. Usually, the accumulated result is expected to be the access latency to a particular memory or device component. Without any additional directions the approach would not be able to deduce such common value. Therefore, the requesting optimization technique has to specify which path is the relevant one for a particular request. This is done by specifying a desired scenario. Currently, there is the option to choose between best-case and worst-case scenarios. According to the scenario a handler will perform different computations. Furthermore, the scenario is used to select among several sub node values the one best matching that scenario. This effectively forms the requested path within the access tree.

3.4.1 Aspect Definition

Aspects in terms of this system-modeling approach basically consist of two items. On the one side, there is the aspect value which represents the current state of an aspect. On the other side, there is the aspect handler which performs computations on aspect values. Figure 3.10 depicts the aspect-handling relations within the system-modeling approach.

Aspect values are any kind of structured data. For example, they can be single value items of numerical type, strings or any type of complex elements. Due to the tight relation to the MACCv2 Framework the aspect-value representation is defined as an object of a C++ class derived from a common aspect representation class. For strings and various numeric value types, a set of predefined classes exists. Aspect handlers are basically computation methods encapsulated into a corresponding object class. A common base class exists which provides the interface definition. The framework ensures invocation of these methods on each system-model entity being part of an access tree. Since the same class of handlers and same value representation can be used for various aspects, an explicit link between the two is needed. Therefore, similar to local properties, each system-wide property represented as an aspect value has a unique name. Aspect handlers are associated with components and channels under this common name.

Accesses and aspects are closely linked together. Since the basic data structure used for the system-wide property computation in terms of aspects is the access tree as defined in the previous section, the query of aspect values is performed based on such an access definition. In general, multiple aspect values can be requested at the same time.

Performing an aspect-value request requires:

- An access definition according to Section 3.3.3
- The set of names for the aspect values which are requested.
- Name of the interesting scenario. Including the aspect name for which the scenario is being taken into account.
- Initiating the query.

After a query has been completed, the root access will provide the combined value for each aspect. Furthermore, the access tree beneath this root access is preserved. Therefore, further analysis of resulting values is possible. Since partial access values are also annotated at these sub-accesses, in-depth analysis of the contribution of each node is possible.

From a formal point of view the aspect-value definition can be represented in terms of an algebraic structure.

$$(A, (f_i))$$

Where A is the domain consisting of the set of values representable in an aspect-value implementation. f_i denotes the operations defined on this domain. Within the aspect-modeling approach, the set of operations is equivalent to the set of aspect handlers for a particular aspect-value type. Due to the intended flexibility of the aspect-value representation and handler implementation, in general no further classification of the algebraic structure is possible. Nevertheless, choosing a particular definition of aspect values (i.e. integral numbers) will lead to more restricted algebraic structures (i.e. commutative rings, in case of integral numbers and aspect handlers performing only additions and multiplications.)

3.4.2 Value Composition

As a prerequisite for the aspect-value composition, an access tree has to be constructed. This happens as described in detail in Section 3.3.4 by following the mapping rules and recording each translation performed. Based on a bottom up traversal, corresponding handlers are called on each component and each channel. The bottom up traversal enables these components to take already computed sub-values into account. This includes the case where cross aspect dependencies exist. An example is the energy consumption of a processing unit. While a processing unit stalls waiting for a memory access to complete it continues to consume a non-negligible amount of energy. Therefore, the latency of the memory subsystem has to be taken into account while computing the energy-consumption value of this example processing unit.

To simplify the usage of this request-based approach, usually the requester is interested in a unique value representing the overall result. Due to the dynamic behavior in typical systems, this unique value may be computed only under the assumption of a selected operation scenario. The scenario is annotated to the request. If required, an aspect handler at a component or channel may take it into account while performing computation of its contribution. Furthermore, it determines in the case of multiple sub nodes in the access tree which one is selected to contribute to upstream values. In the simple case of a numeric value, the operation reduces to either minimum or maximum computation.

Listing 3.1 summarizes the value computation approach as a pseudo algorithm. To keep the representation concise, the construction of the access tree has been encapsulated into a function call according to the approach present in Section 3.3.4. On calling this function the root access has to be prepared as required. Besides address ranges, size and modes, the preparation includes definition of requested value names. This set of requested values is propagated to child nodes in the tree construction process.

Basically, iteration over the tree is performed in an inverse breadth-first approach. For each depth, starting at the highest level, the related node set is collected. Within that node set aspect handlers for each value are executed on each component or channel. Passing the scenario selection argument is required, since the aspect handler may direct its result computation according to selected scenario. Due to this iteration order, the approach ensures for each node currently visited, that all child nodes have already been visited before. Therefore, the aspect handlers may rely on the presence of sub-values.

For reference the parameters are as follows:

- Initial address range: CPU:[0x1000-0x5FF0]
- Alignment: 1 location (each location)
- Size: 16 locations
- Type: Data read

Picking up the access tree construction example described in Section 3.3.4 the value computation is shown next for the same system description and request preconditions.

```
procedure computeAspectValues( Access root,
                               Name scenario,
                               Name scenario_aspect )

  AccessTree T = createAccessTree(root)
  int d = getMaximumDepth(T)
  NodeSet N = getNodesAtDepth(T,d)

  while (N not empty) do
    for each Node n in NodeSet N do
      for each AspectValue v in Node n do

        callAspectHandlerForValue( n,
                                   v,
                                   scenario,
                                   scenaro_aspect )

      done
    done

    d = d - 1;
    N = getNodesAtDepth(T,d)

  done
end
```

Listing 3.1: Aspect value computation algorithm

This example assumes the user being interested in getting energy-related worst-case values for energy consumption and latency. Furthermore, assuming each channel and each component in the system description has associated both required aspect handlers to compute the energy and latency values. For the sake of simplicity, aspect handlers assumed for this example do not consider complex real-life energy-consumption and latency values, but are set to fixed values of 1 nJ or 1 cycle per access. The shared memory has 10 times higher values (10 nJ / 10 cycles).

According to Figure 3.11 the values are computed bottom up. The iteration rounds are indicated in the figure.

1. The first iteration touches the L2 memory component. According to the access tree, there are in total 8 accesses to be processed (due to space limitation only 4 of them are depicted). Main-memory aspect handler computes a contribution of 10 nJ and 10 cycles per access. Since the memory is a leaf node, there is no contribution from sub-accesses to be considered. Therefore, the total value annotated to the access sequence related to the L2 memory remains unchanged at (10 nJ,10 cycles) each. Since all nodes at that level have been processed, the approach proceeds to the next higher level.

2. At that level two sequence nodes exist. Each node encapsulates a sequence of 4 accesses to match each 8 byte wide request on a 2 byte wide bus. Both sequences are calculated the same way. Bus overhead of 4 accesses (4* 1 nJ, 4* 1 cycle) is

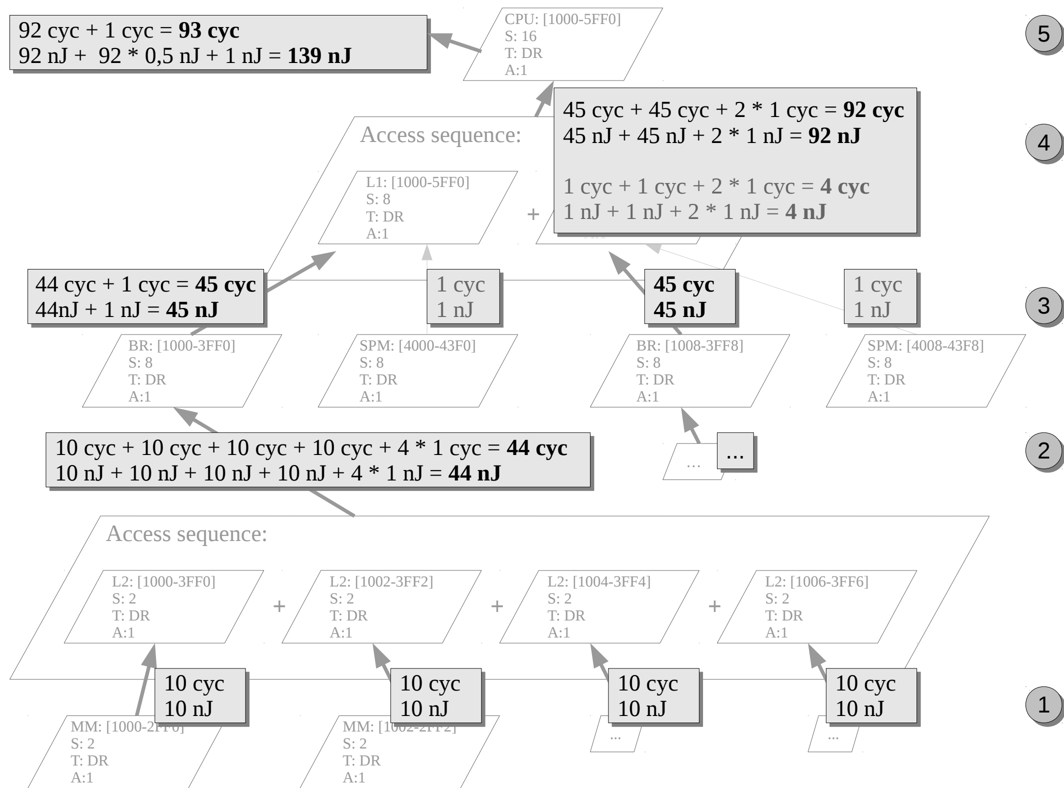


Figure 3.11: Aspect-value computation example.

calculated and combined with the values from lower level, resulting in a total of (44 nJ, 44 cycles) each.

3. Continuing at the next level the contribution of bridges, as well as of local scratch-pad memories, is calculated. Bridges report the accumulated values and their own contribution, in total (45 nJ, 45 cycles). The local memory is quite fast and reports to be able to process requested access within 1 cycle (1 nJ, 1 cycle).

4. At the next level there is again an access sequence. This one has become necessary, since the 16 bytes request coming from upper level has to be split into two requests of 8 bytes each to match the bus width at that level. This level also depicts the situation where several values are calculated for each requested aspect. Since each sub-access may target either the bus bridge or the local memory, there are among other combinations the worst-case accumulated values according to the bridge targeting paths of (92 nJ, 92 cycles) and the best-case values resulting in both sub-accesses targeting the SPM (4 nJ, 4 cycles). Since in this example the worst-case scenario according to energy consumption is of interest, the (92 nJ, 92 cycles) values are reported to upper level.

5. Finally evaluation reaches the root processing-unit component node. At this processing unit, a single access exists, the one provided as request. The aspect handlers associated with the CPU are a bit more sophisticated. Since a processing unit usually does not completely turn off power while waiting for the memory subsystem to provide a data word, it consumes different amounts of energy depending on the latency of the memory subsystem. Therefore, the aspect handler takes this into account by computing a hypothetical contribution of 1 nJ + 92 x 0,5 nJ for the assumed energy consumption in the idle state. The latency overhead is again 1 cycle. Therefore, a contribution of (47 nJ, 1 cycle) is computed. Finally, considering the sub-access, a total value of (139 nJ, 93 cycles) for the worst-case energy consumption and latency is provided to the requester.

Even though the energy-consumption and the latency computation sketched in this example are based on simple arithmetical operations, this is not a limiting factor of the aspect-handling concept presented in this thesis. On the one side, the value representation has been abstracted from a fixed type value to an object-oriented representation. This enables any kind of data types to be suitable for aspect-value representation. Furthermore, the encapsulation of value computation into aspect handlers which are effectively capable of performing any data manipulation, enables other types of algebra to be applied as well.

3.5 Framework

The framework has been designed with implementation effort separation in mind. Variously experienced developers are expected to provide their contribution independently. In a subsequent step these contributions can be combined to form a particular optimization toolchain instance. Figure 3.12 summarizes the three major efforts found in a common source-level optimization technique development processes. Two of them provide building blocks or services, while the third one combines them and implements the actual optimization technique. The roles are not fixed with respect to a long term development process. Besides analysis and transformation techniques which were deliberately designed as service contributions, the previously developed optimization techniques may serve as building blocks for upcoming developments as well.

Taking a close look at the three major development efforts, the first concentrates on provision of system models. Several tasks are required, starting with the definition of participating components and channels. Based on these items a complete system template according to the platform structure has to be provided. Since optimization techniques often require a cost model, the system designer should provide such information via corresponding aspect handlers.

The second group collects analysis and transformation techniques. This group bundles the effort required for provision of libraries of analysis and transformation techniques. An example is the set of analysis technique which are based on profiling (i.e. access count estimation)

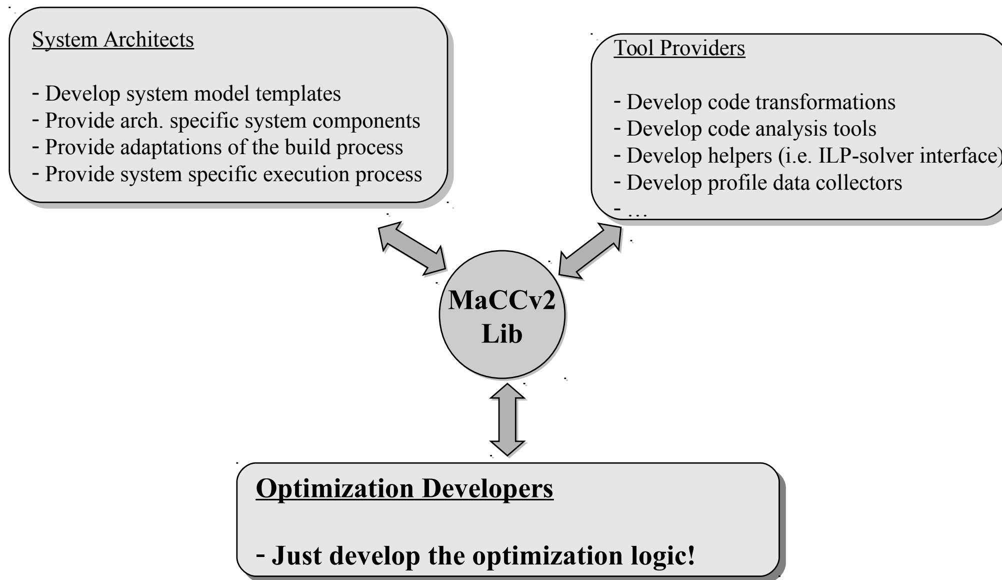


Figure 3.12: MACCv2 usage scenarios.

Finally, the third group refers to the actual optimization technique development process. Here, the framework user applies the techniques developed in the second group and implements the actual optimization technique algorithms which acquire their platform-relevant data from the results provided by the first group.

As can be observed in Figure 3.12, the MACCv2 framework, and especially the core library therein, is the glue to bring these various, mostly independently-performed, developments together.

This library is the hub for any MACCv2 framework-based optimization and analysis techniques. The library bundles the core services provided within this framework:

- A target-system-model representation
- An ICD-C-based source-code representation
- Optimization and analysis step integration methods
- A common-base-class model with reflection support
- Methods for persistent data retention
- User-interface abstraction and a basic text-mode user-interface implementation
- Dynamic library management

Around this central library a set of basic services is provided which complements the library and provides a foundation for rapid source-level optimization technique development:

- Several system descriptions (i.e. MPARM system model, the host system model, etc.)
- Graphical user-interface implementation based on Qt libraries.
- An Eclipse-IDE-based plug-in which provides a graphical system-modeling tool.
- A generic source-code optimization tool, which serves both, for practical use as well as an optimization technique implementation example.
- A set of executables for commonly occurring tasks: Template-based system-description creation, optimization technique invocation and configuration.

With respect to design specification, the processing-step integration has several aspects which are presented in subsequent section. For the supplementary set of service only a short overview is presented here. A more detailed, implementation-related, description is given in subsequent Chapter 4.

3.5.1 Processing-Step Integration

This section introduces the approach for processing-step integration present in the MACCv2 framework. A framework which targets optimization technique development needs to support common optimization implementation approaches. Dividing optimization approaches into a sequence of steps is a common practice in current optimization technique development processes. According to Figure 3.13, many optimization approaches can be naturally divided into:

- An analysis step
- A decision step
- A transformation step

These steps step may be divided into several sub steps again. Typically, this reduces the per-step complexity and enables easier reuse of these steps in related source-code optimization approaches.

Analysis steps are intended for collection of relevant input data for subsequent decision steps. For example, in the context of memory-aware optimization techniques such analysis results could contain access counts to variables. Further analyses may contribute alias analysis results or construction of application-wide data-flow graphs. Different scenarios of interdependencies exist between analysis steps. Either fully independent operation is possible or preceding analyses are needed. In any case, a sub-step-wise structure of the analysis phase can be concluded.

Once all relevant input data has been collected, the actual optimization step can be performed. Here again, sub steps may become advantageous. In general, a good practice, which is also promoted within this framework, is the separation of actual optimization

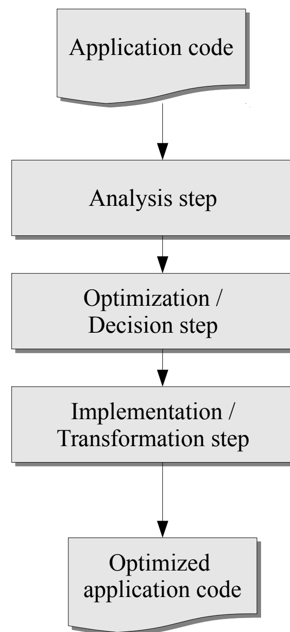


Figure 3.13: Typical optimization steps.

logic, the "optimization decision" step, and the source-code manipulation required to implement these optimization decisions, the "optimization transformation" step. The rationale behind this separation is the increased reuse of a present optimization technique for future platforms. In general only the application-code transformation step will need some modification due to adapting to a different execution environment. In combination with the appropriate set of framework services this advantage comes at only slightly increased overhead. Primarily, this overhead is due to the implementation of a clean and well-defined interface between both steps. In general this is an acceptable overhead, since well-defined interfaces can be assumed as a valuable precondition for successful reuse of optimization techniques.

The motivating optimization technique structure presented up to now exposes a linear step sequence. This is typical for model-driven approaches (i.e. integer-linear-programming-based approaches). Nevertheless, once heuristic or randomized approaches (i.e. evolutionary algorithm-based approaches) are going to be applied, at least after the final code manipulation step, some evaluation of achieved optimization gain or eventually a degradation detection is required. Based on this evaluation result, the optimization technique may be repeated under different input conditions. This evaluation and reapplication of optimization techniques introduces loops into the optimization step sequence. Such loops prevent list-based execution of optimization steps. Instead, some higher-level decision instance is required to direct the optimization step flow and construct conditional loops.

According to the typical optimization technique structure, a conclusion indicates that a framework which supports development of such techniques, needs to provide a foundation for coordinated step-wise execution of analysis, transformation and optimization steps. Since typically such steps may depend on the execution results of other steps, the dependencies between processing steps have to be captured as well. This information will ensure properly ordered execution of such steps. Furthermore, organizing processing steps as a set of sub-steps and the need for conditional execution of steps, including repeated execution, can be tackled best in a hierarchical approach. Therefore, the framework provides means of processing-step invocation within other higher-level processing steps.

The desired scenario for a particular optimization technique or complete toolflow is the seamless reuse for various target platforms, ideally, even for those which were not known at development time of a particular optimization technique. The step-wise approach, presented here, is also beneficial for such versatile reuse scenarios. Typically, a limited set of steps needs to operate in a platform-dependent way. For example, collecting access counts to application-code variables can be performed according to different methods (i.e. profiling or static analysis-based). Furthermore, different target platforms may need different setups or annotations. Without further support from the framework, a toolflow developer needs to specify exactly the envisioned access count collection approach. This effectively requires the toolflow to be modified for each target platform. The MACCv2 framework proposed in this thesis implements automatic processing-step specialization methods based on a classification hierarchy. In respect to the access count example within a hypothetical toolflow, using such access count computation, the request for a generic access count computation step can be specified. The framework will choose the best suitable one at runtime, according to the system model and the available set of approach implementations.

According to these use cases, the details of the processing-step representation approach proposed in this thesis are presented next.

3.5.1.1 Tool Model

Implementing optimization techniques based on a framework reduces the implementation effort significantly. To achieve a sustained implementation efficiency, the reuse of previously developed optimization and analysis techniques is the key. A framework promoting fast development cycles, should support such reuse. One major prerequisite is a well-defined notion of self-contained processing steps. Especially, invocation and configuration of such processing steps needs to be at focus. Furthermore, typically processing steps will not operate in isolation. Constructing toolflows introduces ordering dependencies. Capturing such dependencies in the tool definition, enables the framework to check and try to automatically satisfy such relations.

This section introduces the tool model present in the MACCv2 framework. First, starting from the implementer perspective, tool definition and typical processing-step implementation requirements are presented. Ensuring appropriate tool interaction, a

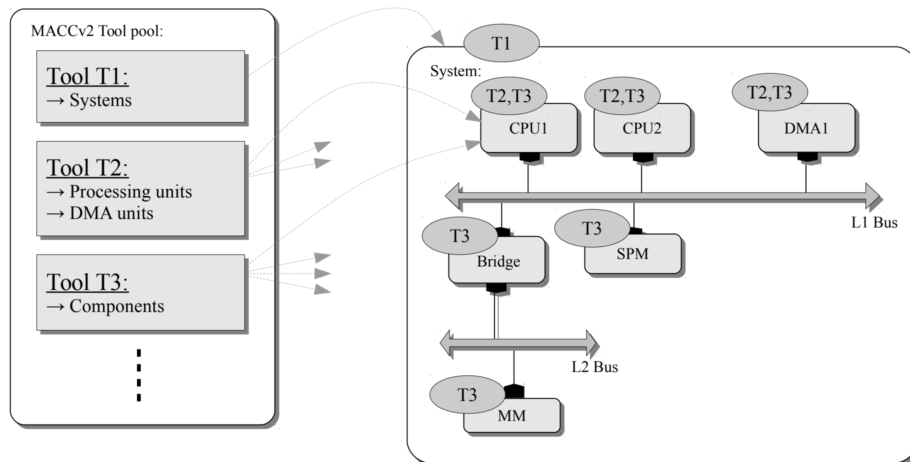


Figure 3.14: Tool applicability.

tool-dependency tracing approach is described next. Finally, the tool model as exposed to the user is presented. in this case, focus is directed towards invocation and configuration.

Tool Definition

An optimization or analysis step encapsulated as a MACCv2 framework tool, is implemented as an object instance derived from a common tool-object base class. The general rationale within this model is to apply such tools to system-model objects. Figure 3.14 depicts this scenario. Each tool defines a set of system-description classes it is applicable to. Since in terms of this framework such objects reflect target-platform entities, this approach matches well the intuitive notion of performing actions for a particular platform component. In the most common case, especially when implementing memory-aware source-level optimizations, a tool will be defined as applicable to processing units. This way, relation to particular target-application code is given. For more tool implementation-related properties, refer to Chapter 4.

Tool Specialization and Abstract Tools

The object-oriented representation of processing steps enables dedicated association of related processing steps to a common base class. Across these classes, inheritance hierarchy can be used to express specialization relations. Especially, this way common interfaces can be implemented. Since framework tools are based on the fundamental common-object-class model, the interface definition includes configuration API entries, enabling unified configuration entries for a class of tools. Typically, interface defining classes omit the implementation; in the case of processing-step tools, such interface tools would not be operational. The framework provides means to express this case. A distinction between regular tools and abstract tools is provided.

According to the class hierarchy, identification of processing-step types and corresponding hierarchical dependencies is possible. Based on this foundation, the framework provides methods to automatically choose the most suitable tool for a particular system-model object. This specialization approach is for example widely used for compilation and linking tools. Since these are highly platform-dependent steps, each processing-unit component suggests a suitable specialized tool which is used when compiling code for this processing unit. From the toolflow perspective, this simplifies the code generation process significantly, since regardless of the actual target platform, always using the generic compilation tool can be requested.

Dependency Tracing

Dividing optimization techniques into processing steps introduces interaction between these steps, and consequently interdependencies. Especially, such interdependences imply an execution order of processing steps. Typically, input data requirements impose such dependencies.

Within the MACCv2 framework, dependencies are expressed as per system-model item tool application requirements. A tool specifies in its initialization method which other tools need to be applied to a system-model object before that particular tool can be applied to such an object. According to the reflection approach present in this framework, these dependencies are denoted in terms of tool class names. Therefore, the actual C++ tool class types of prerequisite tools do not have to be known to a particular tool. This is in line with the modularity concept present in MACCv2.

For each system-model item the framework keeps track of all successfully applied tools. Once the next tool is going to be applied, this application history is inspected for missing prerequisite tools. If missing tools are detected, the framework tries to resolve these dependencies by automatically applying these missing tools. Since such tools may have unresolved dependencies by themselves, the whole process is repeated recursively. Typically, the termination condition is either satisfaction of all dependencies or failure of such, due to presence of a dependency to an unknown tool.

Figure 3.15 depicts a hypothetical tool application sequence executed by the MACCv2 framework once the "ExampleTool" is being requested to be applied. In this figure "ExampleTool" depends on some "AnalysisTool" which in turn needs two distinct "DataCollector" tools. These dependencies are resolved step-by-step. The processing units these tools are going to be applied to have different precondition situation. None of these tools have been applied to CPU1. One of these hypothetical data collector tools have already been applied to CPU2. Finally, CPU3 has got all prerequisite tool requirements already satisfied in some preceding run. Once dependencies have been resolved, correlated tool sequences are applied to each processing unit.

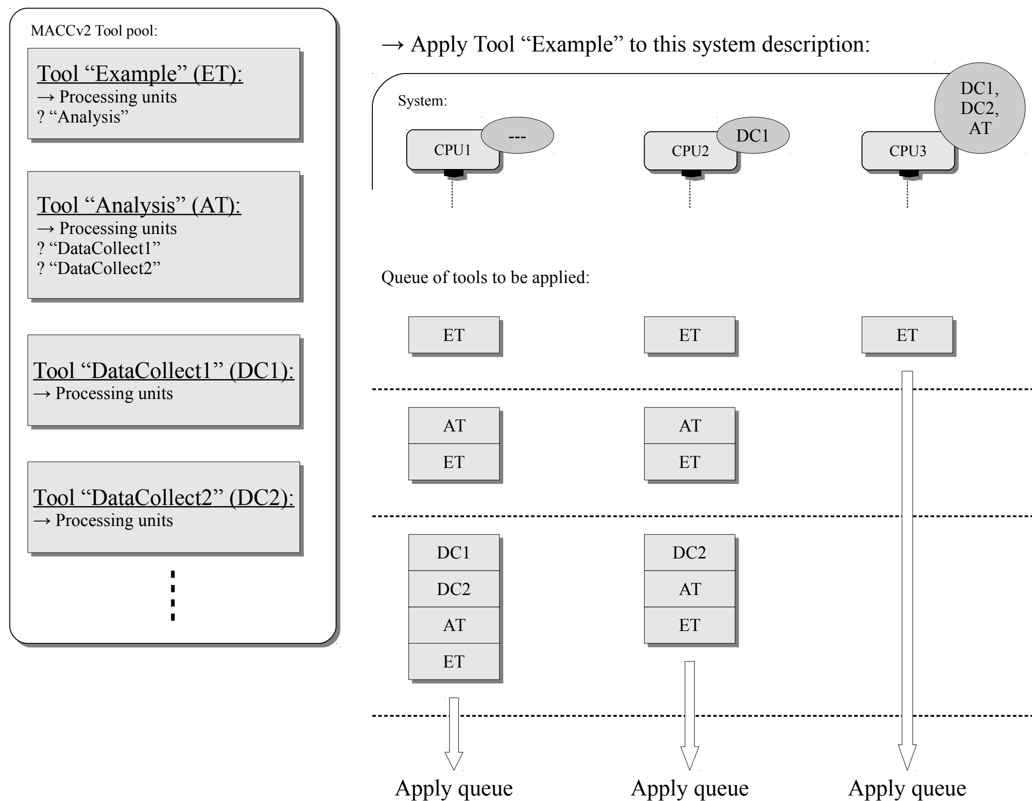


Figure 3.15: Tool-dependency-driven application sequence.

3.5.1.2 Processing-Step Interaction

Except for the basic case where processing steps perform only modification to the system description or to the application code, typically cooperating processing steps need to pass information to each other.

The framework does not impose any limitations on the way such communication can be performed. Nevertheless, the preferred communication approach is based on the common-object-class model presented in Section 4.2.1. Passing information from one processing step to another is preferably done via object annotations. As described in the referred section, this approach is superior to others in the context of the MACCv2 framework. Especially, the seamless integration with the system description, type save access and minimal effort for persistent data retention are key benefits.

Focusing on the framework-based tool interaction, in general passing processing results from one tool to another is performed via a system-description-attached object graphs. The object classes, the attachment location, the number of graph instances and the naming conventions are defined by the optimization technique.

Besides passing the actual optimization technique results between framework tools, a coarse-grained control over the operation of a particular processing-step tool is often

required. Typically, such control parameters will vary at different scope levels. To support such variable configuration options for any kind of tools, the framework offers methods for definition of so-called tool settings, which are associated to system-model objects. Based on these tool settings, the framework offers several supporting methods to create, manipulate and find tool settings feasible for a particular tool. To provide a well-defined, and descriptive interface, tool settings expose their entries via framework-based configuration API.

Further, tools may also need some operating system and runtime-environment-related configuration options. These configuration options are quite constant. They are best expressed as MACCv2 environment variables. Such environment variables resemble the same well-known approaches in Unix-based operating systems. Actually, among various types of such environment variables the framework provides means to link operating-system environment variables to corresponding ones within the framework, enabling configuration of framework tools from some invocation script.

Concluding tool operation configuration options, there are basically four places :

- Tool-settings objects attached to particular components of the system model.
- Configuration fields in the default tool settings object.
- Direct configuration fields in the tool object.
- MACCv2 environment variables.

Summarizing the processing-step integration in general, the definition of tool interfaces and configuration options, provides the tool integration foundation. Tool interfaces are the preferred method for passing processing-step results between tools, while tool settings and the direct configuration entries, provide control over the operation modes present in a given tool. Typically, a toolflow tool exposes only settings and configuration options of the topmost processing step to the user, enabling user interaction in a well-defined way, regardless of the actual user-interface type.

3.5.2 Supplementary Framework Services

The MACCv2 framework offers a versatile set of fundamental services which is available to optimization technique developers, as well as being used for implementation of upper-level service within this framework. In particular these are, the system description, code representation and processing-step representation presented in previous sections. A short overview of these fundamental services is presented next. A detailed description can be found in the subsequent implementation Chapter 4.

The integration facilities provided by a framework already need support at a very basic level. Especially, a streamlined API across all kind of services is a valuable precondition. Fundamental to such an API is a common notion of objects. Several programming languages (i.e. the most prominent one is Java) provide natively such a common base class for any kind of derived object classes. The C++ language, identified as frequently

used for optimization technique development, does not. Therefore, the framework compensates for the missing common-object notion and defines in the context of MACCv2 framework such a base class model. It is expected to be used as the root class for any kind of object classes. It is defined as `IR_PersistentObject`. Especially, the ICD-C-based application-code representation, the system-modeling approach and further framework services exploit this feature excessively. The common base class of this framework provides several additional features:

Persistent data retention and serialization support: A serialization API for the common base class and the corresponding derived classes is provided. This way object graphs can be stored to persistent storage and reconstructed later in another runtime context. The implementation effort in derived classes reduces to enumeration of relevant class members.

Reflection and Versioning: In complex, runtime-pluggable implementations, compile-time-unknown object class may occur. Therefore, examining the class hierarchy is often very useful. This service provides support for object-class naming and identification. Since object-class implementations may vary, version numbers have been introduced to help identify whether the expected implementation of a class is available.

Runtime linking: Since modularity is an high priority goal in this framework, low-level support is needed to achieve this goal. Typically, modules are represented as operating-system-related libraries, which can be dynamically linked into current execution context. Support within this framework focuses on identification and location of such libraries, as well as provision of an object-class name to library mapping.

User-data annotation: Any class derived from `IR_PersistentObject` supports the annotation of so-called user data. User data are named references to other objects of this common class or any derived class. These annotations can be used to attach addition data to any object or create cross relations between distinct object. This kind of annotation is commonly used for analysis-step results storage at related application-code representation items.

Auto deletion, reference counting and pointer tracing: Object auto deletion and reference counting services help in keeping a clean process image even in complex data structures. Corresponding objects may keep track of references and eventually self destruct, once not needed anymore.

Object-Configuration API: The configuration API is closely related to the reflection services. The configuration API provides means to expose a subset of class member variables in a uniform way to other objects. These members can be examined or modified without having access to the C++-related object-class declaration.

Object Events: Object events provide means of notification across object hierarchies. Objects within this framework may post notifications to other objects in case some relevant event or state change has occurred.

Object-Container API: Object containers are basically named collections of object references. They are primarily used to construct dynamic data structures.

Factories: Factories take the basic object construction methods provided in context of persistent storage a step further. Additional configuration parameters can be specified to direct the object-instance creation. Factories are typically used in construction of new target-platform system-description instances.

Next, fundamental service offered within this framework is related to the runtime environment. Executing optimization techniques on a particular host system typically requires adjustment of several runtime parameters. Especially, locations of external tools (i.e. compilers) may vary among different host system platforms. An often-used approach to cope with such variations in the execution environment is based on environment variables. The MACCv2 framework offers an API to define such variables in an object-oriented way. Beyond plain operating-system-related environment variables, more sophisticated types are defined. Examples are, counter variables or unique identifier generators.

Further, according to the goal of keeping as much recurring and infrastructural work away from optimization technique developers, this framework offers a user-interface abstraction service. Optimization techniques face a quite abstract user-interface API which allows for implementing typical user-interaction tasks in a runtime-environment-independent way. This way the same optimization technique implementation can be used for presentation purposes along a graphical user interface as well as deeply embedded into an automated optimization toolchain with no user interaction at all.

These typical user-interaction services are offered within the MACCv2 framework.

Progress indication and logging: Typically, complex optimization techniques require prolonged execution times. To enable progress tracking and presentation, any optimization technique may provide progress indications. Even in the case of toolchains, the framework ensures appropriate accumulation of these values. This gives a good execution time estimation to the user.

Dialog-based interaction: In some cases, optimization techniques may require direct user interaction in terms of providing some runtime values or choosing among several options. In this case, the MACCv2 framework offers an API which allows for formulation of such requests in a concise way and user-interface-independent way.

Object views: Especially in the case where novel optimization techniques are presented to the public, it is beneficial to display optimization technique decisions and final results in a graphical way.

3.6 Conclusion

This chapter presented in detail the key approaches and models proposed in this thesis. Starting from the system-modeling approach, over the calculation of system-wide properties based on aspect handling, up to the MACCv2 framework which embed a rich set of supplementary services for rapid and easy memory-aware optimization technique development.

Contribution of a System-Modeling Approach

In accordance to the general goal of this thesis, the system-modeling approach aims at applicability in the context of development of source-level optimization techniques. Even though these techniques operate at the abstract source level, the knowledge of architectural properties is beneficial or even vital for this type of optimization techniques. In contrast to simulation-based approaches where the full-system execution is at focus, the system-modeling approach presented here focuses on request-based, database-like provision of target-platform properties. A balanced approach between initial modeling overhead and level of details has been achieved.

Primary motivation for this system-modeling approach was the observed need for a common target-platform description in the often occurring multi-step-based source-level optimization and transformation technique implementations. Common assumptions on the target platform help to avoid optimization decisions which interfere with subsequent steps.

According to the intended application scenario, the system-modeling approach provides a model at the processor-memory-switch (PMS) level. The general structure consists of components representing self-contained entities in the target platform and channels representing the interconnects between them. These items are defined by a unique class name plus a variable set of properties attached to them. This locally focused structure enables development of libraries containing sets of components and channels. These libraries are usually structured according to the target platforms being modeled.

Locally-scoped properties attached to components have a major draw back. Due to their scope, they contain values related only to that particular component or channel. Especially, drawing system-wide conclusions from such data sets is difficult. Therefore, an approach to retrieve such system-wide properties has been proposed. It covers the most common case where such properties are computed according to the perspective of a processing unit. System-wide values for energy consumption and latency related to accesses performed by processing units have been a driving example throughout this part of the chapter. In terms of this system-modeling approach these system-wide properties are called aspects. Each aspect is represented by a value type and a set of handlers operating on values of such a type. Requesting such aspect values is performed based on accesses. Accesses are used to construct access trees which capture every possible access path in the system model for a particular request. These trees are used to determine the invocation order of aspect handlers to compute a combined system-wide property value.

The combination of a structural model incorporating component and channel properties with an access-oriented, request-based, system-wide property computation method, results in a unique system-modeling approach. The advantages of such an approach are twofold. On the user side, fast iteration over a structural model is possible as well as uniform, request-based, exploration of processor centric system-wide properties. On the system-model developer side, the system-model design is greatly simplified. Each component or channel can be regarded as a self-contained entity. Combining these components or altering their properties is simple, since the local scope avoids occurrence of global side effects.

Contribution of an Optimization Technique Development Framework

Developing source-code optimization techniques is a tedious task. Recurring implementations, marginal reusability, mostly due to unknown or misleading interfaces, different assumptions on the target-platform properties are only a subset of problems occurring in the typical development process of source-code optimization techniques.

The MACCv2 framework targets these problems and provides a common foundation for a wide range of optimization and transformation techniques. Even though the framework is not limited to the combination of ICD-C-based precise application source-code representation and proposed multicore-aware target-platform representation, both features render it preferably suitable for implementing memory-aware optimization techniques for MPSoC platforms.

The framework focuses on increased reuse of optimization techniques. One very basic but key foundation which increases the chances to successfully reuse an optimization or analysis technique in upcoming developments, is a common notion of data representation and a set of methods to retrieve and store such data. Since this framework targets optimization technique development implemented in C++ programming language, data representations are naturally encapsulated in C++ objects. Such an object-oriented representation has additional advantages. For example, object classification and inheritance relations help to improve the structure.

Based on this common foundation almost all other framework services use and benefit from such unified object representation. Namely, persistent data retention, flexible object annotation, system model and source-code representation, processing-step encapsulation and several more presented in this chapter.

Having only a solid low-level foundation would still require recurring implementations for common problems. Typically, application-code representation, a target-platform model and some means of interaction with the user are prevalent tasks. The MACCv2 framework covers all these areas.

These framework services are directed towards a single optimization or transformation technique, which would imply a monolithic approach. Since such monolithic approaches are tailored towards a single application scenario, this is disadvantageous for later reuse. Within this framework a modular approach is supported. The key concept here is to promote the subdivision of optimization or transformation techniques into smaller steps, and if applicable to, do this for these steps hierarchically. In terms of this framework,

such steps are denoted as tools. The framework services cover a whole range of methods for definition of such tools and interaction between them. In particular, classification relations, precedence dependencies, configuration options and interfaces between them can be defined. On the usage side, the framework ensures enumeration and identification of present tools. With respect to tool invocation, the class-based tool definition approach enables automatic tool specialization, which in turn, improves reusability, since sub-related processing steps may be requested in a generic way, with the actual implementation being used varying between toolflow setups or target-platform definitions.

Summarizing the framework-related contribution of this thesis, the design and the set of services provided within this framework can be considered to be a well-balanced foundation for development of memory-aware optimization techniques. As described later in the evaluation part, this framework and the proposed system-modeling approach have shown practical relevance in the MNEMEE project. The set of optimization techniques developed in this project is organized as a multi-step tool chain which benefits from a common target-platform model. Utilization of MACCv2 framework has significantly reduced the effort for integration of optimization and transformation techniques developed at disjoint organizations across Europe.

4 Implementation

4.1 Introduction

Developing source-level optimization and transformation techniques results in a significant amount of recurring work. This thesis proposes the MACCv2 framework which aims at reduction of this recurring overhead by provision of a common foundation for a wide range of optimization and transformation techniques. One key recurring aspect in the development of optimization techniques in general and source-level optimizations in particular is: The development of a target-platform description. A system-modeling approach has been proposed within this thesis. The practical implementation of this system-modeling approach is tightly integrated into this framework.

To cope with the complexity of current optimization techniques, developers tend to implement step-wise approaches which subdivide a task in a better tractable sequence of steps. The framework presented in this thesis plays the coordinating and integrating role in such a scenario. A wide-set services is provided within this framework. The complexity of these services starts at a very basic level which is closely related to the object-oriented C++ programming language, and continues up to complex services as the system description or processing-step representation.

As indicated in the previous chapter, the framework services have been implemented as a set of C++ object-oriented APIs. This fits well with the common approach to develop memory-aware source-level optimization techniques in the C++ language. The application code to be optimized is represented as a set of C++ objects constructing an abstract syntax tree. The C++-based API for this system description, processing step and runtime-data representations have well matching interfaces and C++-based APIs which form a well-defined framework.

According to the concepts and models proposed in this thesis, a closer look at the actual implementation is taken in this chapter. Practical examples of framework services and system-model components accompany this implementation description. The implementation centric perspective of this chapter puts actual APIs and class names at focus. Providing this information is expected to give a good overview to the way framework services have been implemented and serves as a starting point for the development of upcoming optimization techniques or further target-platform descriptions.

The following sections iterate over each service provided within the MACCv2 framework. Starting at the fundamental services related to the common object base class, continuing at higher-level services as the user-interface abstraction and system-model interface, and finishing at the processing-step representation and toolflow construction.

4.2 Framework Services

The MACCv2 framework proposed in this thesis is based on a central component, the MACCv2 library. This library is the hub for any MACCv2 framework-based optimization and analysis technique. The library bundles core services provided within this framework which are presented in detail in this section. Starting from the fundamental common-base-class services which include among others data retention, reflection and annotation services. The implementation of the system-modeling approach and processing-step representation are based on these services. Finally, a close look is taken at the user-interface abstraction services.

4.2.1 Common Base Class

The integration facilities provided by a framework already need support at a very basic level. Especially, a streamlined API across all kinds of services is a valuable precondition. Fundamental to such an API is a common notion of objects. Several programming languages (i.e. the most prominent one is Java) provide natively such a common base class for any kind of derived object classes. The C++ language, identified as frequently used for optimization technique development, does not. Therefore, the framework compensates for the missing common-object notion and defines in the context of MACCv2 framework such a base class model. It is expected to be used as the root class for any kind of object classes. It is defined as `IR_PersistentObject`. Especially, the ICD-C-based application code representation, the system-modeling approach and further framework services exploit this feature excessively. Besides commonly found methods for serialization and reflection, the common base class of this framework provides several additional features which are presented next. Figure 4.1 provides an overview of these features.

4.2.1.1 Persistent Data Retention and Serialization Support

The first common task tackled in this framework is the persistent data retention. Optimization techniques may require preserving their state across multiple runs as well as pass analysis results to the subsequent processing steps. The ad-hoc methods often found in state-of-the art optimization techniques define more or less well-structured text files where such information is stored. Obviously, this comes at the cost of several disadvantages. The file format is not well-defined or documented. Subsequent changes may break easily existing toolchains. Non-uniform value representation, ambiguous hierarchical structure, unclear field semantics are only a few problems which can occur while performing text-file-based data retention. Avoiding these problems, even in combination with sophisticated libraries (i.e. XML parser libraries), comes at high effort for infrastructural work and documentation. Furthermore, this has to be repeated for each new data type and optimization technique implementation.

The MACCv2 framework proposed in this thesis defines a serialization API for the common object base class and corresponding derived classes. The effort for derived classes basically reduces to enumeration of members expected to be stored or loaded

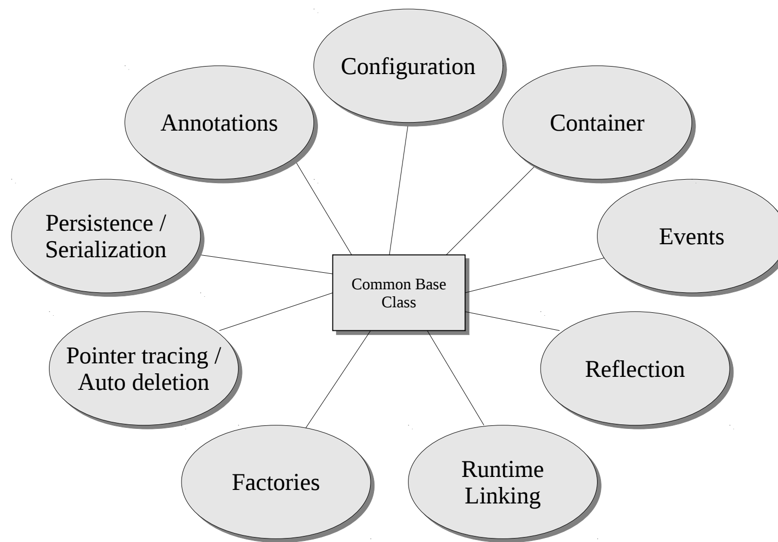


Figure 4.1: Common object features.

in the corresponding methods. Items of commonly occurring data types can be stored and restored this way, including all basic C++ data types, some common STL types and containers (i.e. strings, lists, sets, maps and vectors) and pointers to other object and members which are derived from an IR_PersistentObject class. Especially, this possibility to store cross-object references and complete object hierarchies is a valuable add-on to common serialization approaches which often support only storage of basic data types. The serialization backend within this framework takes care of serialization of all referenced objects, ensuring a complete, self-contained representation. This serialization approach matches well with typical methods for representation of runtime data or analysis results which are going to be passed to subsequent steps. The natural and easily-accessible runtime representation can be preserved with minimal overhead. An example could be a map of symbol references from one application-code representation to another one. Such maps are beneficial if source-code modifications are performed in preparation of profiling runs. Usually, these profiling runs will be performed on modified or annotated application code, while the actual outcome has to be related to the original application code. Such maps encapsulated into a storable persistent object can be made available to a profiling trace analysis step within few lines of code. The entire overhead of storing and reconstructing of such data structures is covered by this framework. Effectively, the analysis and optimization steps can operate on native C++ data structures as they would if no persistence consideration were required.

The actual storage format is not fixed. The MACCv2 framework provides an extensible backend which defines a lean API to the actual file reader and writer implementation. This way various storage formats can be implemented. Currently the framework supports

a binary file format, which is fast and dense, but not human readable and a XML-based file format, which can be manually explored.

4.2.1.2 Reflection and Versioning

The second feature often found in the context of common-base-class implementations is an API reflection method. Examining the class hierarchy is often very useful especially in combination with runtime linking where the set of known object classes is not constant during application runtime.

The reflection approach included in this framework focuses on class hierarchy representation and instantiation of objects. The method for class hierarchy exploration covers retrieval of object-class names, queries for inheritance relations and few class properties (i.e. whether a class is abstract or can be used to construct preinitialized objects).

Classic reflection implementations also provide methods for member enumeration. This implementation follows a slightly different approach. An object-configuration API is provided, which enables controlled enumeration and modifications of selected members. Refer to Section 4.2.1.7 for more details on this service.

Class naming within this approach is based on a string representation of class names as occurring in the C++ source code. A string-based approach has been intentionally chosen, since in combination with runtime linking, the class name is often the only information available, rendering type-based approaches infeasible due to the required knowledge of the actual class declaration. The MACCv2 framework supports object creation based on these class names. The most prominent situation where such an object instantiation is performed, occurs when an optimization technique is going to be applied to a system description. Such techniques are encapsulated into (tool-) objects, which are instantiated according to their requested name.

The framework provides methods to control this generic instantiation of objects. First of all, the class implementer may inhibit such instantiation. In general, a class has to be "initializable“ in terms of this framework to be feasible for instantiation of objects of this class. Basically, a class is in this state if it implements an initialization method which preloads class members accordingly when requested by the framework. The second condition which may inhibit instantiation of a class is if the class is abstract in terms of the C++ language (i.e. it has unimplemented virtual members). Abstract classes are marked as such to the framework, so it will prevent creation of such objects.

Since the framework-based reflection support is an add-on to the native C++ capabilities, the `IR_PersistentObject` derived classes need to provide the required class names and inheritance information. The framework hides this overhead almost completely from the user. Actually, the approach requires only two short lines of code. The first one, is required in the class declaration. The second is placed in the file containing the class implementation.

Within the class declaration a class version number can be specified. Class versioning is primarily used in combination with the persistent store of object hierarchies. For each serialized object the class version is recorded. Once the data is being restored, this version number is compared against the current object class implementation. The framework

will report an error and refuse to restore the data, if a class versions mismatch is detected. Whether a particular class version numbers mismatch can be decided by the class itself. Therefore, an improved class may report compatibility to its current version and to some older version enabling migration paths to more recent optimization techniques.

4.2.1.3 Runtime Linking

Another service provided within this framework is the dynamic loading of libraries containing object class definitions. There are two cases where such runtime linking of additional code is performed. The first case is basically on user-code request, in particular, once creation of an object of an unknown class is requested. In this case the framework tries to locate a dynamic linkable library containing this class definition. There are several locations which are checked for such a library. First of all, the framework tries to determine whether a library file name is known. If so, a library with that name is searched. The mapping between classes and library names is provided at the class definition. For a particular toolchain, a configuration file can be generated which contains the relevant mappings. The framework reads this file on startup and tries to match the library names according to this list. In the case no library can be determined for a particular class, the class name is used as the filename. Looking for such linkable files is performed according the following scheme:

- If a library is known, it may have an explicit path defined, so look-up at this location is performed first.
- The framework has an environment configuration variable defining possible library paths. Searching there is performed next.
- Finally, apply the operating-system search order (i.e. use `LD_LIBRARY_PATH` variable in Unix-based OS).

If a corresponding library is found, it is loaded and the reflection data is updated automatically. A subsequent retry to create an object of requested class will most likely succeed, if the mapping information has been correct.

The second case where runtime linking of additional code is performed occurs while restoring object graphs according to a persistent store data file. While this file is read, corresponding objects are created. The same situation as in the case of a user initiated request may occur: The requested class is currently not known within the framework's class hierarchy. The same search approach is applied as in the manual case. In addition to the manual case, the persistent data contains an additional library hint, which may be used if no suitable entry in the class name to library map has been found.

This dynamic runtime loading in combination with the serialization approach provided within this framework has a unique advantage compared to other state-of-the-art data retention approaches, effectively enabling a combined data and code retention. The possibility to bind member methods to a particular data set and this way providing for example getter/setter methods has several advantages over any kind of plain data

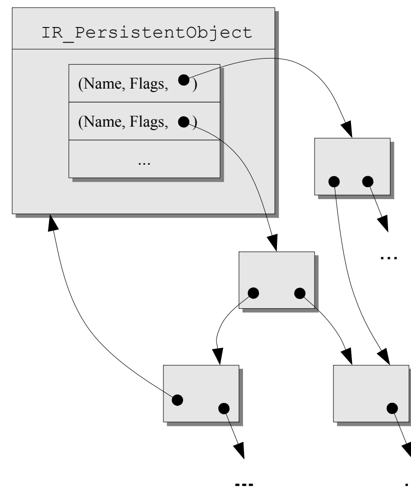


Figure 4.2: Example of an annotated object graph.

retention (i.e. enabling sanity checks, access control or basically any computation on this data is possible). Intuitively spoken, from the optimization technique perspective the de-serialized data may be considered to be "smart" since computation methods are made accessible instead of plain read/write access to it.

Since runtime linking modifies the process image of an optimization technique, basically arbitrary code may be inserted. In open environments this may be hostile code, too. To provide a protection against unwanted behavior, the framework provides methods to the running process which can be used to check a particular library file in advance. Which kind of checks is performed is user-dependent. From simple name checking up to signature-based check, any method is possible. The framework will load a library only if the check callback has returned an approving answer.

4.2.1.4 User-Data Annotation

The common base class provided within this framework defines another valuable feature for combining analysis and optimization techniques into automatic toolflows. Any class derived from `IR_PersistentObject` supports the annotation of so-called user data. User data are named references to other objects of class `IR_PersistentObject` or any derived class. Names are arbitrary string constants. Each name references a single object. Figure 4.2 depicts an example of this relation in a object graph. Besides methods for direct association and retrieval of user-data objects, wild-card-based look-up methods are provided. Such methods can be used to collect similar objects in terms of names or object classes. Especially, the possibility to restrict the set of retrieved object references to a particular class simplifies the iteration code within an optimization technique. Since the retrieved set-members are correctly typed, no casting and validity checks are required.

Since referenced objects are only required to be derived from this common base class, almost any data can be annotated. For common cases where a numeric value or a plain string is needed, predefined container classes are provided. Furthermore, the ICD-C-based application-code representation as well as most other services of the MACCv2 framework, especially the system-model representation, are based on this common-base-class model. This enables any of these entities to be referenced in the user data. Further, any of these objects will accept user-data annotations. A simple example demonstrates how powerful these annotations are. Let us assume an allocation technique, which assigns each global variable to a particular memory. Instead of generating some map-file which links variable names to memory names, this user-data annotation allows for definition of a plain pointer to the targeted memory component which is attached to the global variable representation. Especially in the case of MPSoCs such map-files may run into ambiguity problems when the same code is being executed on several cores, while this annotation-based approach is always precise. In combination with the serialization approach presented in the previous subsection, this annotation is as persistent as a plain, file-based, map would be.

Associating object references is accompanied with a set of flags which provide hints to the framework and eventually to the optimization technique on each association. The current set of flags covers primarily control over automatic actions performed by the framework. In particular this set of flags is implemented:

- User-data references may be marked with a flag indicating to the framework that the object referenced in this entry has to be deleted once the object holding the reference is destructed.
- Another flag operates in a similar way. In contrast to the previous one, even the removal of such a user-data entry implies the deletion of the corresponding object. This flag saves implementation effort, since the user does not have to retrieve the reference and invoke the deletion process.
- A flag may also indicate to the framework that the removal of this user-data association is required when the referenced object is deleted. The framework provides pointer tracing services, which are used here to get notified once the referenced object is deleted.
- Similar to previous flag, this one also uses pointer tracing to get notion of the referenced object being deleted. In contrast to the previous flag, the implemented behavior is more rigorous. Assuming the object holding the reference is useless or invalid without that particular user-data annotation, it will be deleted as well.
- User-data entries can be marked with a flag which identifies the data to be of temporary nature. While serializing an object to a file, user-data entries marked with this flag are ignored.
- Finally, there are currently 8 predefined flags which are not used by the framework. They may be used by optimization techniques to indicate some special states or

conditions related to a particular user-data entry. The semantics of each flags is user-defined, and will most likely depend on the type of associated object and needs to be known a priori by any optimization using such flags.

Optimization techniques may have different assumptions on the set of associated user data. Especially, they may request user-data entries which are not present. In the case that an optimization technique is not enumerating present user-data entries via wildcard-based methods, but is requesting a single item with an exact name, the MACCv2 framework tries to construct such an entry, if not present in the user-data set. The framework provides methods to register callback functions which will be invoked whenever a request for a particular user-data entry could not be satisfied. Within this callback function the requested entry may be created. If successful, the requested entry will be returned to the optimization technique.

An application scenario where such on the fly entry creation is beneficial, could be a computationally intensive analysis technique. Instead of requiring precomputation of every analysis result in advance, only these actually requested ones will be computed on-demand. Since such on-demand results are attached as regular user data, no multiple recomputation of repeatedly requested user-data entries is required.

4.2.1.5 Pointer Tracing

In complex data structures, mutual cross references between objects are the common case. As long as the application code has knowledge of all places where a reference to an object is kept, cleanup of such reference points can be performed before a particular object is deleted. In complex optimization techniques which consist of several libraries - most likely contributed from various sources - this knowledge is not trivial. Therefore, often various ad-hoc update and notification mechanisms are introduced to tackle this problem. The framework proposed here benefits from the definition of a common base class. It implements a unified approach for pointer tracing and invalidation.

There are two types of references to be handled by this approach. The first type refers to object-references associated via user-data annotations. The framework takes care of such annotations automatically, if the corresponding flags (refer to Section 4.2.1.4) are set. Basically, two types of actions can be performed: Removing just the reference or deleting the entire object holding this user-data reference.

The second type of references are plain class members. To be able to identify invalidation of references stored in local member variables, a callback mechanism has been chosen. Derived classes may implement a callback method to cleanup the reported references and eventually invoke its base class implementation to continue the cleanup process. In contrast to smart-pointer approaches (i.e. provided by the BOOST library [61]) this approach does not introduce additional wrapper objects around pointer members, resulting in a reduced object size with no additional access overhead. Especially, in combination with arrays of references a significantly lower space and runtime overhead is expected.

The notification mechanism allows for group notifications. The callback method is invoked with two arguments passed. The first is the group name, the second is the

actual pointer becoming invalid. Invalidating according to a group name is useful if validity of multiple references is required to be kept synchronized to provide the expected functionality. Group names are plain character strings and can be arbitrarily chosen by the referencing object class. The framework provides methods to associate one or multiple object references with a particular group name.

4.2.1.6 Auto-Deletion and Reference Counting

Pointer tracing is a valuable service to keep a sane overall data structure state if the optimization technique requires explicit object deletion. Another commonly-found practice defers object deletion until the last reference has been removed. This resembles aspect of automatic garbage collection known from other high-level languages (i.e. Java). The MACCv2 framework covers also this programming paradigm for object classes based on the common-base-class definition. Objects which will keep track of the number of references and perform auto destruction, once no more references are open, are denoted as "attachable" objects. Especially in context of user-data attachment, the framework keeps track of reference count updates, so no further user code overhead and interaction is required.

The framework supports two approaches for reference counting and automatic object deletion. The first is suitable in the case of an entire class of objects which require reference counting. The second one is beneficial if the majority of objects of a particular class does not require reference counting, but few instances do. In this case reference counting can be enabled at per-object level.

For class-based attachable objects, the framework provides a base class denoted as `IR_AttachableObject`. It is an immediate descendant of the common base class named `IR_PersistentObject`. It inherits all base class features, plus additional reference counting support. Object classes which should be recognized by the framework as attachable have to be derived from this more specialized class.

The second approach providing reference counting is based on attach guards. Currently, the framework implements this approach as user-data-related sub-objects which perform the reference counting and deletion handling. The framework recognizes these specific user-data object references and treats the referencing objects the same way as the one based on an attachable object class. Per-object reference counting is commonly used when dealing with ICD-C-based code representation objects. Assume an optimization technique which needs to preserve several implementation variants for a particular statement (i.e. a loop). Such alternative implementations could be attached as loop statement object references to the original loop representation being part of the application-code syntax tree (AST). The original implementation has a well-defined and known dependency in the AST, therefore can be cleaned up properly within ICD-C without reference counting. The additional implementation variants are not known to ICD-C and would remain unreferenced in memory. Marking such loop statement objects as attachable, will prevent such memory leaks.

Attachable objects are most beneficial in combination with user-data-based referencing. This way the framework can take care of updating the reference counts accordingly.

```
IMPL_CONTAINER_CONFIG_BEGIN(MyConfigTest,MACC_Container)

  ADD_CONFIG_ENTRY(boolcfg,"Some␣description")

  ADD_CONFIG_ENTRY_ENUM_BEGIN(intcfg,"Another␣description")
    ADD_ENUM_VALUE("hello",33)
    ADD_ENUM_VALUE("world",99)

  ADD_CONFIG_ENTRY_ENUM_END()

IMPL_CONTAINER_CONFIG_END()
```

Listing 4.1: Configuration definition example

Nevertheless, the framework provides an API for manual control of reference counts. Similar to pointer traces, once objects are referenced in regular class members the referencing object has to inform the framework about the reference taken. Doing so can be encapsulated in a setter method. Basically, two methods are provided by the framework. One for increasing reference counts and the other for decreasing.

4.2.1.7 Object-Configuration API

The framework presented here promotes a plug-and-play approach. This implies the provision of self-contained libraries which implement services at various levels. In such a case, the reflection mechanism is valuable once it comes to examine the set of exposed object classes and instantiation of corresponding objects. At the object-level scope some insight into the object structure is useful. The framework supports exposing selected member variables via a uniform interface. In terms of this framework this service is denoted as object-configuration API. A motivating example for such configurable objects are system template factories. The properties of a system description being created can be configured in advance. For example, properties exposed by the MPARM system template factory are the number of cores or memory sizes.

The framework provides fine-grained control over the set of exposed members. This control covers the type of access or the set of valid values. Exposing member variables can be done in quite a concise manner. The framework provides a rich set of macros which reduce the effort for the developer to enumerate the set of member variables to be accessible as configuration entries. For each member variable, a developer may chose whether it is a read-only variable or a variable which may be modified. Further, the user-visible name has to be provided (usually the same as the variable name) and optionally a short description giving the purpose of this configuration entry. Listing 4.1 shows an example of a configuration definition for an exemplary object class `MyConfigTest`. This definition can be placed along the usual class implementation into the same file. Besides this implementation block, the class declaration has to contain a `DECL_CONFIG` line announcing the presence of configuration entries.

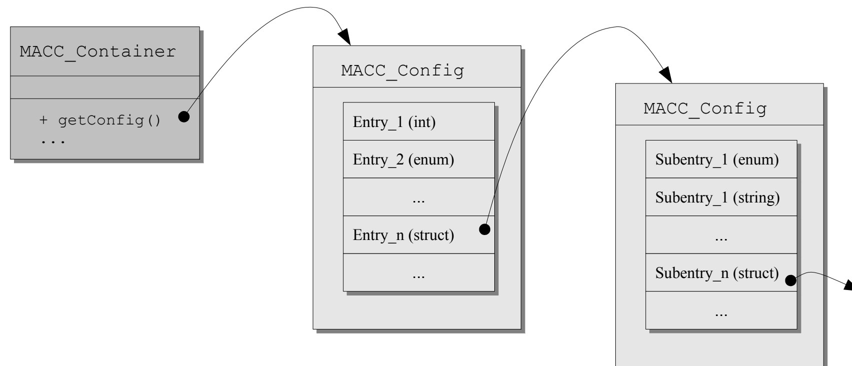


Figure 4.3: Configuration object structure example.

Restricting the set of acceptable values for a read-write entry can be done either by explicit enumeration of these values, or definition of a valid value range in case of a numeric entry.

There are several other configurable options for each entry. For example update notification callbacks can be registered for each entry, or visibility levels defined.

Besides a rich set of common numerical and string entry types, several framework-specific configuration entry types can be defined. First of all, the configuration API allows for construction of hierarchical configuration trees. Entries which point to member objects which themselves are capable of providing a configuration tree can be defined. A graphical user interface exposing the configuration entries to the user will recognize such sub-tree structure and present it in an adequate way. The second set of framework-specific entries covers various data types provided by the MACCv2 framework. Examples are a unit-equipped real number representation or environment variable representations. Defining such entries does not differ from entries of simple data types.

From the configuration user perspective, the framework provides unified methods for accessing any kind of configuration entries. The general structure is as follows: The object exposing member configuration entries provides a method which returns a reference to the collection of configuration entries of this object. Figure 4.3 depicts an exemplary structure. In general, configuration entry values are exposed in a string-based representation. For simplified access to these values, corresponding setter/getter methods are provided. Accessing a particular configuration entry is done based on its entry name. For a more detailed access to a particular entry, a reference to the configuration entry object has to be requested first. Once this reference is known, additional information can be retrieved. This includes the set of valid values or the value range, description text, type information or error reports, in the case an invalid value has been provided in a call to the setter method.

Both access methods have a preferred application scenario. While the name-based method would be preferably used in text-based or automatic setups, the object-based

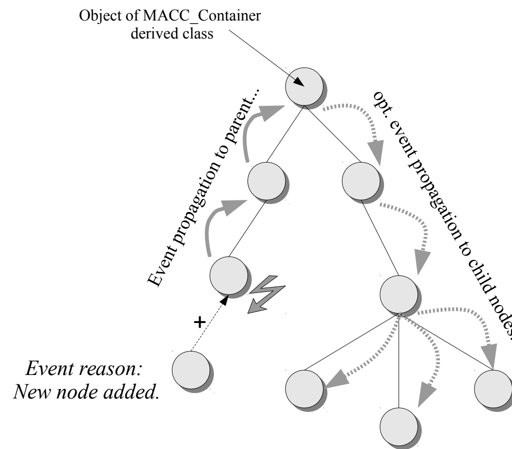


Figure 4.4: Object event notification example.

method is valuable for graphical interfaces where detailed information can be exposed to the user.

4.2.1.8 Object Events

Changes to individual objects may have effects on various other objects in complex data structures like the system descriptions presented in this thesis. An often occurring example in this context would be the modification of application code which may invalidate computed analysis results. The low-level consistency ensuring approaches previously presented are not applicable here, since on such modification, the semantical information is invalidated and not the structural properties of a system description. Therefore, another approach is required to cover such situations.

This framework provides an object-based event notification mechanism. Events are similar to the well-known C++ exceptions. In contrast to exceptions, events do not propagate along the current call-stack, but they use parent-child relation between objects to propagate within a given object graph.

An event in terms of this framework is an object instance of a particular event class. Common to all event classes is the requirement to be derived from a common event base class. An object which has undergone some relevant change which is expected to influence other objects, may initiate an event notification. In general, a suitable event object, which reflects the occurred change will be instantiated and posted to the framework service for distribution. The framework distributes the event object until a receiver method indicates the event object as being consumed, or no more distribution paths could be enumerated. Figure 4.4 depicts the event distribution in case of object modifications due to item addition to a related object container.

```
IMPL_EVENT_HANDLER_BEGIN(MyEventTest,MACC_Container)

  ON_EVENT(MACC_Event_Modified)
  {
    ev->getSomeData();
    ev->doSomething();

    return EVH_CONTINUE;  // go on
    // return EVH_BREAK;  // or stop, if consumed here
  }

  ON_EVENT(MACC_Event) // catch any event...
  {
    // ...
    return EVH_CONTINUE;
  }

IMPL_EVENT_HANDLER_END()
```

Listing 4.2: Event handler implementation

Objects which have to be sensitive on event notifications have to provide an event receiver method. The MACCv2 framework supports implementing event receiver methods by provision of a set of macros, which basically reduce the effort for the developer to provide the actual action to be performed on arrival of each relevant class of events. Since events are based on the regular C++ class hierarchy, requesting sensitivity on a generic event class also covers all derived event types. Listing 4.2 depicts an event handler implementation. The handler reacts to object-container modifications in the first section, and basically to any other event in the second section. Since the event actions are evaluated in the order of occurrence, this is usually the preferred implementation order to achieve the intended behavior. More specific event types have to precede more generic ones.

Once an event has been created, the framework has to propagate it to all relevant objects. For each event, the class and the scope of relevant objects can be defined. Object scopes are defined according to a parent-child relation. In general, in this context objects referenced in user-data entries are considered to be children of the referencing object. Since objects can be referenced at several places, the opposite relation has to be explicitly specified. When using the object-container API presented in the next section, the framework creates these relations automatically. Events record the set of objects already notified. Therefore, circular references can be avoided. Both explicit parent definition and circular reference detection make this approach applicable not only to object trees but also to object graphs. An event definition advises the framework to use a particular scope of propagation. Following notification schemata are available:

- The first one performs basically no propagation. Only the event handler at the object causing this event is involved. Self notification can be useful if the developer choses to implement object-state-change processing in a centralized place.

- The second type propagates events only to child objects (i.e. to objects referenced in the user-data entries)

- Another propagation type advises the framework to notify the parent object and follow up this relation until no parent reference exists.

- This propagation type combines the previous three types. First the causing object is notified, then the child objects, and finally the parent objects.

- A system-level notification is performed starting at the root parent node down to all child nodes reachable from that point.

- Finally, a global notification of all objects reachable via the child relation from the global MACCv2 runtime-environment object is performed. The main purpose of this propagation method is to notify objects related to different system descriptions.

4.2.1.9 Object-Container API

Managing complex data structures involves recurring implementations. Iterating over data collections, inserting, removing or searching for subsets in such collections are always occurring tasks. In the context of this framework, the system-description implementation has exposed properties of such complex data structures. At an abstract level, the system description consists of various types of object instances. A hierarchical dependency between these objects is given due the structure of the system description. Starting at the system level, a system keeps collections of components and channels. A component keeps track of the corresponding ports. Finally, ports need to collect the related mapping rules. These examples and several more occurring in the system description would require implementation of similar APIs and corresponding behavior for different object types. The approach presented here does not focus on implementing the next most efficient data collection method. Efficient approaches exist (i.e. STL containers) which are not duplicated here. What is usually missing, and causes tedious overhead in the development process, are user-friendly interfaces to such data collections. Typical implementations consist of a private container object and a set of content-related methods which basically proxy the modification request to the private container object.

The MACCv2 framework provides the building blocks for such container API. The effort for the user is limited to one declaration line, which primarily specifies the object type to be stored in such a container. Optionally, differing names for the private container and method naming can be specified. Such a declaration expands to a set of methods which provide the common access patterns. The method names are adapted according to the user-provided declaration (i.e. declaring a container API for objects of class “Port“ will provide among others a method ”AddPort” which adds new objects to this container).

In particular the API includes following methods:

- Add a new object reference to the container.

- Remove a particular object reference.
- Remove a subset of object references according to naming or tagging criteria.
- Retrieve a subset of object references which satisfy the specified class type, naming and tagging criteria.

Especially, the last method is a powerful look-up and enumeration method. The possibility to ensure presence of only properly typed objects in the result set saves significant casting and filtering overhead in user code. A common example is the enumeration of a particular type of components of a given system description (i.e. only processor components). Obtaining such data basically requires two lines of code:

```
map<string, MACC_Processor*> processors;
sys->getComponents("*","*",processors);
```

Where the "*" indicate no restriction on names or the set of tags.

The object-container API supports two types of object storage. The first one uses a regular member variable of type `MACC_Container`, the second method keeps the container as an attached user-data item. Both storage approaches have their specific advantages. The member variable-based approach is faster, since one less level of dereferencing is required to access the data. The second approach is advantageous in case the stored items have to be enumerated in a uniform way. Since the container is part of the user-data set, an external entity (i.e. a graphical user interface) would be capable of enumerating such a container and its content without knowledge of the actual class declaration, solely via framework-provided interfaces.

The framework-provided container class is derived from the common object class. Therefore, such container object inherits all properties of this class. Especially, the serialization is useful for persistent retention of container content. In addition to the base class properties, the container class adds primarily methods related to implementation of object-container API. This includes, container tagging support, extended search and enumeration methods for user-data items and a uniform object naming interface.

4.2.1.10 Factories

Continuing the set of framework services related to the common object base class, object factories are presented in this section. Factories extend the class-name-based object creation method provided within the reflection support. Factories are beneficial if more control over the type, initialization state or the instantiation of full object structures is required. The main application scenario for factories is the construction of system-model items and complete system descriptions.

The factory service incorporated within this framework provides a set of methods to enumerate available factories. To get an overview of available facilities, a method which returns the complete set of factory objects is provided. Further methods are provided for requesting a factory with a particular name or to retrieve a specific factory which is capable of constructing objects of requested class.

Once factory references are known to the user, further details can be enumerated. Each factory provides the set of class names of which it is capable to construct objects. Typically, a factory will have a set of configuration entries to control its mode of operation. These configuration entries can be enumerated and used according to the methods described in Section 4.2.1.7.

A typical process of constructing objects via framework-provided factories consists of the following three steps:

1. Request the factory object capable of constructing a particular object class.
2. Setup the factory configuration entries accordingly.
3. Invoke the object instantiation method.

The first practical examples of factory-based object instantiation can be found in the graphical system-model editor. A user may add new components and channels to the system model. First, the set of available component or channel classes is enumerated from available factories. Once the user chooses to add a component, this set is presented in a list. Choosing a particular item from such a list results in the instantiation of a corresponding object via the corresponding system-model factory.

A second example uses the configuration capabilities excessively. The dedicated toolchain frontend developed for the MNEMEE project uses the factory approach to instantiate fullfledged system models according to predefined templates. The toolflow user may configure the number of cores, memory sizes or any other platform-specific option before the system description is created according to these parameters.

4.2.2 Code Representation

In the context of this framework, the previously available ICD-C intermediate representation has been slightly extended. Basically, the major goal was to make ICD-C benefit from the advantages of a common-object-class model. Therefore, the ICD-C intermediate representation has been integrated with the common-base-class approach presented in Section 4.2.1. Due to this integration, a common foundation across the system-modeling implementation, the application-code representation and the processing-step integration approach exists. Such a cross domain foundation is a subtle, but very beneficial property of the MACCv2 framework. Expression of optimization-technique-dependent cross-object relations and annotations is possible with only minimal effort for the optimization technique developer.

4.2.3 Runtime Environment

Executing optimization techniques on a particular host system typically requires adjustment of several runtime parameters. Especially, locations of external tools (i.e. compilers) may vary among different host system platforms.

An often-used approach to cope with such variations in the execution environment is based on environment variables. Such variables are preinitialized in the setup phase according to current execution environment. Later, the content is used when path strings are constructed (i.e. when constructing the command line to invoke a particular compiler). The MACCv2 framework exposes a similar service. Environment variables are represented as pairs associating a name with the corresponding value. Based on such tuples, which have an object-oriented representation within this framework, methods are provided for searching, iterating over and modifying these environment variables. Beside this object-oriented API, string replacement is supported within the environment variable API. Basically, user-provided character strings are passed to the framework for substitution of variable names with their values. Similar to the operating-system-based variable substitution, undefined variables are substituted by an empty string.

Since framework-provided environment variables are represented as object instance, this object-oriented representation is especially advantageous in the context of string substitution. In contrast to the operating-system-based environment variable implementation, the associated value does not have to be constant. The environment variable object may implement a value computation method, which will be invoked each time the variable value is requested. Typical examples of non-constant environment variables provided within the framework are random character sequence generators or counter variables. Since such variables are resolved in the same step as any other environment variable, they can be used in a straightforward approach for path construction of temporary filenames.

Currently four classes of environment variables are provided within this framework:

- Plain static environment variables, defined within the framework context.
- Host system bound environment variables. These reflect and update the values of related operating-system-provided variables.
- Random character sequence generator variables. The variable value is a configurable number of random characters (typically 8).
- A unique number generator. Environment variables of this class return an 64 bit integer number. The number is incremented on each value request. The unique value is globally shared among all variables of this class and preserved across multiple process invocations.

Due to the typically flat name space for environment variables at the operating-system level, excessive usage of such variables may result in conflicts or at least in a cumbersome representation of the runtime environment. The framework offers a hierarchical approach for environment variable representation. According to Figure 4.5, starting at a single global environment, several subenvironments can be defined. Typically, these subenvironments are related to particular processing-step implementations or system-model representations. An optimization technique or an object within a particular system-model representation holds a reference to its specific environment, if any. In general, if no specific environment can be determined, the global one is referenced.

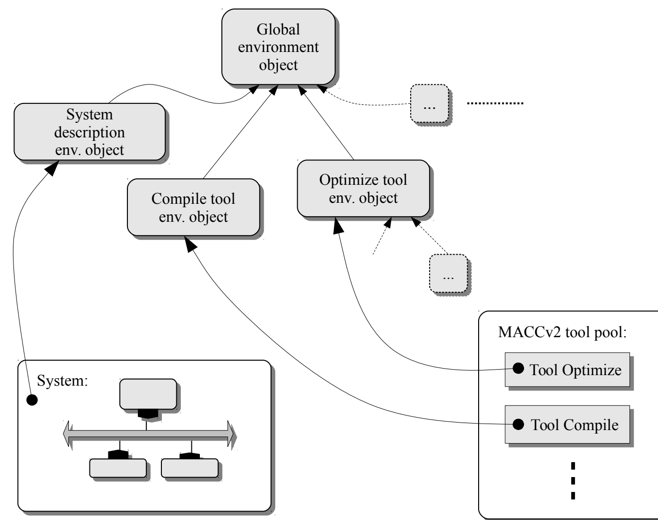


Figure 4.5: Environment structure

Across these runtime environments a parent-child relation is defined. Except for the topmost global environment, each one references a parental runtime environment. This parent-child relation is used once environment variables are going to be resolved. Environment variables are resolved starting at the immediately referenced environment. If some variables do not exist in this context, the parent environment is consulted.

Adapting optimization techniques to a particular runtime environment is supported within this framework. Based on the well-known approach of using environment variables to setup runtime-variant configuration options (i.e. paths to external executables, paths to the workspace etc.) a similar service is provided within this framework. From the optimization technique perspective, these runtime environments are persistent even across multiple runs in subsequent processes.

Since the framework targets support for development of highly reusable optimization techniques, a toolflow typically exposes a fine-grained modular structure. Self-contained parts (i.e. processing steps, system-model components) are encapsulated within dedicated libraries, which are linked together at runtime. The framework keeps track of several parameters for each of such libraries. For a typical library, the most important parameters are the set of provided object classes and the file system path from where the library has been loaded. The first list is helpful to find the mapping between a particular object and the corresponding library. Once such a relation is found, the library path can be retrieved. Assuming a well-known folder structure relative to the library path, all related files can be accessed without any installation or setup phase, even when the library and its related files are moved within the file system.

In addition to the already loaded libraries, the framework keeps, in a similar way, track of a user-defined set of expected libraries. Such sets can be defined in the toolflow construction phase. Let us assume a complex optimization technique is going to be

delivered for a predefined target platform in context of a project. In this case, the set of required processing steps and the system-description type are well-known in advance. According to this knowledge, a list of required libraries can be constructed. During the startup phase, the framework consults this “libcache” file for the set of libraries and eventually the paths where to search for them. Each library is loaded into the process image in this phase. This way failures due to broken or missing libraries can be detected early in the toolflow runtime.

Keeping track of the set of libraries is an important service within this framework. Especially since their filesystem location can be used to reference related files without the knowledge of their absolute path.

4.2.4 User-Interface Abstraction

Complex optimization or analysis techniques typically require prolonged time to perform their task. Especially in a multi-step toolflow these times accumulate to a significant amount. If such processing steps are not applied as some automatic background processes, but explicitly invoked by a user, operation times higher than a few seconds without any feedback will most likely cause the user to suspect a failure. To avoid such situations, it is beneficial to keep the user notified of current actions being performed, and eventually estimate the expected duration. The MACCv2 framework provides such progress indication and logging services to the optimization technique developer.

A second set of services related to user interfaces provided within this framework focuses on implementation of interactive modes of operation. The optimization technique developer may decide to involve the user into the optimization process by requesting some input, asking to choose an option from a set of choices or displaying intermediate results. The MACCv2 framework supports this set of interaction methods.

Since these methods need to be fully independent of the type of interface exposed to the user, the general approach is to provide a quite abstract API, designed in a dialog-oriented way. The optimization technique prepares an interaction request and advises the framework to expose it to the user. Once the user has performed the requested interaction, the result is passed to the optimization technique and its operation continues. In the context of graphical user interfaces this resembles a modal dialog box.

Besides these basic interaction methods, the user-interface API manages so-called object views. Each user-interface implementation may provide presentation and editing methods for dedicated object classes. An optimization technique may request an object view for a particular object instance. Basically, any object instance of a class which is derived from the common object class, can be a suitable target. The MACCv2 framework tries to match a compatible view provided by the current user interface to the requested object. Since these views are not required to be modal, they are suitable for intermediate state presentation while the optimization technique continues to perform its task.

The user-interface abstraction tries to keep easy and efficient reuse of optimization and analysis techniques always at focus. Especially, strict decoupling of a particular user-interface implementation and the API provided to the optimization technique gives

the freedom of choice for the intended usage of an optimization technique. The same implementation, and even the same binary library encapsulating a particular optimization or analysis technique, can be used for both extreme situations; on the one side deeply integrated into an automatic workflow, where no user interaction is required, and on the other side for presentation purposes, where interaction, and intermediate results should be nicely presented to the audience.

The subsequent sections give a more detailed overview of capabilities provided by each interaction method.

4.2.4.1 Progress Indication and Logging

Keeping the user informed about the current state of a processing step avoids impatience or even frustration in the case of prolonged operation times.

The framework provides basically two types of services related to user information. First of all, there are logging and error reporting methods. Further, there are progress indication methods.

The logging and error reporting methods provide basically a line-oriented interface. Reporting a line of text is similar to printing text via the well-known formating “printf” method. The framework offers four output channels:

Info Output directed to this channel is for reporting processing-step states and general information on the action being performed. In general, the reported text should be concise for this channel.

Log This is basically another informational channel. According to typical logging of processing actions, the content directed to this channel should be more detailed. Longer text, even spreading across several lines, is acceptable here. The output should give a good overview of what is going on in the optimization or analysis step. Eventually, this even includes some overview of achieved results (i.e. estimated energy consumption or runtime reduction)

Warn Reporting of any non-fatal problems, missed conditions or any state which may, but does not have to, harm proper operation of the optimization or analysis technique should be directed to this channel.

Error Finally, any fatal failure, misconfiguration, or invalid operation should be reported here. Typically, any condition reported via the error channel is assumed to be a possible source of a non-successful termination of the optimization or analysis technique.

To further classify the output generated by an optimization or analysis tool and give the user control over the level of detail presented, each reported text line is annotated with a verbosity level. The verbosity level is an integer number, with the least verbose messages indicated with level 0. The user may choose a maximum verbosity level. The framework ensures that only messages with an indicated verbosity equal or below this threshold are presented to the user.

4.2.4.2 Dialog-Based Interaction

When designing interactive tools, typically not only output to the user is required, but also requesting input from a user may become necessary. Currently, the MACCv2 framework supports two methods of requesting such input from the user. The first asks the user to enter an arbitrary line of text. The second method provides a question and offers a set of choices to the user. The user is expected to select one of them.

Since the framework abstracts from the actual user-interface implementation, no assumption on the way both input methods are presented to the user are made. The interfaces are basically straightforward text-based interfaces. For the text input method, the optimization tool developer needs to specify a caption text, some description explaining the purpose of this input and eventually some default text, which may be proposed to the user as possible reply.

In the case of a selection method, similar items as for the text input method need to be specified. In addition to the caption and description, a set of choices needs to be passed to the framework. Each choice is represented as a tuple consisting of a text associated with this choice, an optional description, which may be shown in GUI-based interfaces as a tool-tip pop-up, a unique id and eventually a pointer to additional optimization-technique-related data. To reduce implementation overhead, the framework provides several sets of predefined answers. They cover typical “Yes-No”, “OK-Cancel” etc. cases.

Integrating user interaction into an optimization or analysis tool also has a down side. Typically, this prevents script-based background operation of the same implementation of a particular optimization tool. The framework presented in this thesis targets this problem at two levels. In general, any user-interface implementation has a flag indicating whether an interactive operation is desired. This flag may be cleared to indicate a background operation mode without any interaction. The optimization technique can query this flag and avoid requesting user input. Clearly, this moves the burden of taking care of non-interactive operation to the optimization developer. If a developer chooses not to take care of the operation mode, the framework still tries to ensure uninterrupted background operation. If non-interactive mode is chosen, the input and selection method do not show any user interaction. Instead, the default answer text is returned in case of an input method, or the first choice is selected in the case of a selection query. At this level, the only effort for the tool developer is to consider appropriate order of answers, or provide a default input text, which ensures desired operation.

4.2.4.3 Object Views

Simple dialog-based interaction is suitable for requesting provision of a small number of decisions. Especially, once it comes to representation of some intermediate results, solely dialog-based interaction combined with line-oriented text output will not deliver the expected user experience. Therefore, an object-based user-interaction approach has been integrated into this framework.

The approach is depicted in Figure 4.6. In contrast to the dialog-based user interface, where plain-text messages are exchanged, this one operates on objects. In particular, the

example contains several object-view instances for the currently selected user-interface implementation. Each view indicates which type of object it can handle. Object-class hierarchy is considered as well. Based on this, the fundamental idea is to request the framework to provide an opaque, compatible to the current user interface, representation for a given object. A processing-step implementation decides on the type of representation to be requested according to given tags and name. If a suitable object representation has been found, the corresponding handle is returned to the processing step. In a particular example, requesting a handle for the COMET-System object will result in a handle to the generic system view, since no more specialized implementation exists. In contrast to this, the MPARM System has a dedicated view, which would be provided in that case. Typically, views are requested on processing-step results. This can be observed in this example as well. For the taskgraph construction results, which have been attached to a MPARM CPU, a dedicated view exists, while the access count results do not have any dedicated view. In this case the most generic view for the basic `MACC_Container` object class is returned, if present.

Once such a handle has been obtained, this handle can be used to trigger several actions. Basically, the processing-step implementation controls the visibility and content updates of this representation. Further, if a modal operation mode is desired, the processing-step implementation can postpone its own execution until the representation has been closed by the user. Any further exchange of data, or control of operation is performed via the object being viewed. This reduced interface combines well the flexibility needed to represent complex data structures or create more sophisticated user interactions with the requirement not to depend on a particular type of user interface.

Since the operation of an object view and the processing-step implementation's main thread of execution can run concurrently, without any further synchronization, there would be a great chance of failure or at least of displaying of invalid object states. Therefore, shared access is regulated via an update method. In general the object being viewed is accessible to the processing-step implementation. The view representation is expected to operate on a snapshot of the object's content. Such snapshots are generated on initialization and each time the processing-step implementation calls the update method. Therefore, the processing-step implementation needs to ensure a consistent state of the object only while the update method is executed.

Due to the decoupled approach of using object views which are developed independently of the processing-step implementation, increased emphasis in the MACCv2 framework has been put on the method used to enable the requesting processing-step implementation to identify a particular view. Currently, three types of views are distinguished along the level of interactivity they offer. Least interactive are plain static view. They display the state of the associated object without any user interaction. The second type denotes browsers. This type of object views allows for some user interaction. Selecting items to be more closely inspected or integrating different view perspectives are only two examples of possible interaction. Finally, the third type allows for modification of associated object state. This type allows for changing object properties or modifying and creating object references between sub-related objects.

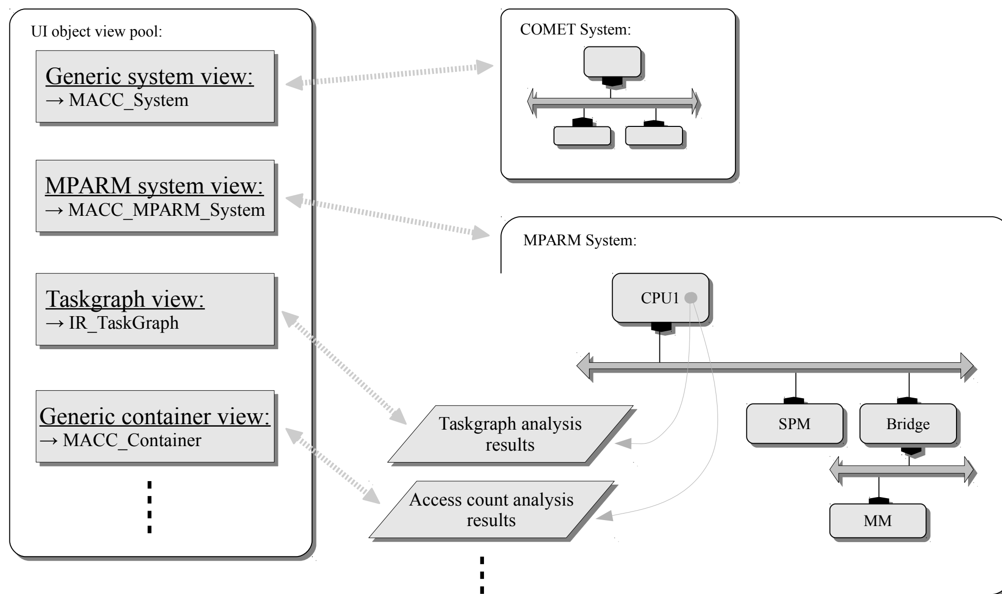


Figure 4.6: Object-view relations

Besides these type definitions, other tags can be defined as well. Once requesting a view, the framework checks if the view has the requested tags defined. Which views are examined for the set of tags depends on the object compatibility. Each view defines in its initialization phase which objects classes it can handle. The class identification is based on the reflection approach introduced withing this framework. Therefore inheritance relations can also be considered. The processing-step implementation may request providing views which match exactly the object class or enable the framework to consider also base class views. Using base class views has several advantages. The first advantage targets view reuse. Since derived object classes can be viewed by their base class representation future processing steps are likely to benefit from present view implementations. In this context such views will probably be less intuitive, or will not contain all details, but at least user interaction can be satisfied. Another benefit of base class views enables simple construction of object-graph browsers. Such graph browsers will chose to rely on automatic selection of most suitable view type for each node. Providing a generic view for the common-persistent-object root class can be used as a fallback mechanism in such a browser enabling it to display any type of object nodes.

Two object views incorporated into the Qt-based graphical user-interface implementation are presented as practical examples next. The first provides a browser for HTML documents. The second is an image viewer. Both are typically used to represent processing-step result summaries, or corresponding diagrams.

According to the general object-view approach, these views also operate in an object-centric manner. To represent HTML text, the framework defines an object class. This

MACC_HtmlData class is a container for the source code of a HTML page. In addition to the storage location, this class of objects offers few editing methods (i.e. a “printf()” like interface) for constructing and modifying the HTML content. Furthermore, since typical HTML documents reference other documents, the storage object may contain sub HTML-Data objects which incorporate referenced pages. A processing step intending to expose a HTML document to the user instantiates such a HTML-Data object hierarchy, fills-in the content, and requests the framework to provide a viewer for the root object. If the currently active user interface provides such an object viewer (i.e. the Qt-based UI), the corresponding handle is returned.

Similar to the HTML view, the image viewer also operates on a particular object class. In this case the expected object class is denoted as MACC_ImageData. The content is basically the binary representation of a PNG image. Since the image data representation is derived from the generic binary data representation, the same interface for accessing the content and file I/O can be used.

Concluding the user-interface abstraction, the framework implements a balanced approach between the abstraction level and the possibility to exploit features of a specific user-interface type. Typical application scenarios are covered. Therein, output-oriented services are: Logging of processing-step output, error reporting or progress indication. Simple user interaction can be achieved via text input requests and selection dialogs. Finally, more sophisticated user interaction can be achieved via an object-viewer interface. Exposing processing-step-specific information based on dedicated object types allows for use of all techniques available for a particular user-interface type (i.e. a GUI-based on the Qt library) to represent the content of such an object in a user-friendly way.

4.2.5 Practical User-Interface Implementations

Several user-interface implementations are provided within this framework. Mainly dependent on the intended application scenario a particular type of user interface is best suitable. To cover the typical range, three different types of user interfaces are provided. There is a plain-text-based interface implementation, a graphical user-interface implementation which uses the Qt framework and a user-interface implementation which integrates the MACCv2 framework into the Eclipse-IDE. Each of them is presented in detail in subsequent sections.

4.2.5.1 Text Mode

Integration of processing steps is performed via command-line executables in typical development workflows. Therefore, if MACCv2 framework-based source-code optimizations are going to be part of such workflow, a command-line-based interface is required.

To satisfy this requirement, the framework implements such a plain-text command-line interface. The implementation consists basically of a user-interface abstraction adapted to text-based input/output and a set of executables which can be used to perform typical task (i,e. invocation of processing steps or construction of system descriptions)

The user-interface abstraction consists of several services which need to be provided in a text-based fashion. Please refer to Section 4.2.4 for details on the complete set of services. At this point as a good example the chosen user-interaction approach is presented.

The first service to be presented here provides logging and error reporting facilities to the processing step. A text-based interface matches straightforward with the line-oriented reporting. Each report of an appropriate level is passed immediately to the user terminal. The formating is limited to combining all invocation parameters into a line. A typical logging report looks like this example:

```
L:MNEMEE:Loading system 'mparm/parallelizer/edge-detect.xml.gz'
```

The first character indicates the type of the line. There are four types: **L**ogging, **I**nformation, **W**arnings and **E**rrors. Afterwards, the caption text is appended. This typically indicates which entity reports the line. Finally, the reported text is printed. If the operating system supports colorful terminal output, the lines are reported in different colors according to their type.

The second service provided via the user-interface abstraction offers services to request user input. Here, a text-mode representation for arbitrary text input as well as a selection-based input of predefined options, are provided.

```
?:Create System:Provide a name for the new system:
```

Again, a similar syntax is applied. A question mark indicates the input request, followed by the caption text and the request of what is expected to be entered.

The second input method offers the user a set of options to choose from. An exemplary request may have the following structure:

```
Q:Factory selection:Please choose which factory should be used for...
 1) : MPARM-System Factory
 2) : Host-System Factory
?
```

A **Q**uestion is indicated first, followed by the caption text, and the actual question text. In subsequent lines the possible answers are enumerated. Each is preceded by a sequential number. The user is expected to enter this number when selecting one of these options.

Further user-interface abstraction provides methods to indicate the progress of actions being performed. Since several actions and sub actions can be active simultaneously, the progress reporting is organized in a hierarchical structure. Each entity (i.e. an optimization or analysis tool) reports its local view of its operation progress. The framework combines these local values to an estimated overall progress report. In the text-mode user-interface implementation such progress reports have the following syntax:

```
P: Total/MNEMEE_Toolflow/HTGParallelizer='Performing parallelization...
```

The overall structure can be observed in this example. The **P**rogress report type is indicated in the first character. The hierarchical path of the currently-reported progress is printed next. A short description of currently-performed action follows. Finally, the completion percentage is reported as a progress bar followed by the actual number.

Besides the user-interface abstraction, a set of command-line executables is provided to help integrate MACCv2-based optimization and transformation techniques into other workflows. Most often occurring tasks are tackled in these executables. Namely, setup and configuration of the runtime environment (macc-setenv), construction and modification of system descriptions (macc-createsys and macc-modsys) and invocation of processing steps (macc-tool-apply).

These executables offer a rich set of command-line options enabling their execution without further user interaction. In addition to this mode of operation, they can operate interactively. In this mode, the user is guided verbosely towards the desired action and requested to provide input. Typically, only a single command-line argument is required, which points to the system-description file to operate on.

Using the MACCv2 framework-based processing-step implementations and optimization techniques via text-based command-line interface is subject to a few recurring steps. Typically they are performed in the following order:

1. Setting up a runtime environment for a new workspace is performed via a macc-setenv command. This is typically a one-time effort.

2. Creating a system model for a target platform is performed via macc-createsys. This command can be used to create system descriptions in a single step. Typically, configuration options can be passed to the system factory to control the properties of the target-platform description being constructed.

3. The previously constructed system description can be preserved to be reused for several application-code assignments according to this step. Using macc-modsys the application source code can be associated with a particular processing unit.

4. The actual optimization or transformation technique is invoked via macc-tool-apply.

5. Finally, typically the optimized source code needs to be read back from the system-description file to be accessible in a plain-text format. This can be performed via the macc-modsys command as well.

Except for the first step, which does not require any argument, all other steps in the sequence need at least the file reference to the system-description file to operate on. Due to the well-encapsulated processing-step representation within this framework, the steps shown here are almost universal for any optimization or transformation technique to be performed.

The executables offer an interactive mode with verbose user guidance. Hence, for the sake of brevity only an example of the user interaction for the source-code association step is presented next. The macc-modsys command can be invoked as follows:

```
#> macc-modsys -i -S application_code.c system.xml
```

This command line chooses to use the interactive mode, and specifies the filename of a compilation unit to be added to the system description, indicated in the last argument. Since the set of command-line arguments does not specify the processing unit and the intermediate representation to operate on, this information has to be requested at runtime:

```
Loading system system.xml ... OK
Q:Processor selection:Please choose a processor for modification:
1) : ARM Processor #0
2) : ARM Processor #1
3) : ARM Processor #2
4) : ARM Processor #3
? 1
Selected processor ARM Processor \#0 at system sys1
Q:IR selection:Please choose the IR for modification:
1) : - New -
? 1
?:New IR:Provide IR name:
IR1
Selected IR IR1
Added compilation unit application_code.c
System sys1 has been modified. Saving to system.xml ... OK
```

The command-line-based invocation of processing steps is not very user-convenient compared to the other user-interface types. Nevertheless, this is a frequently used interface type, especially when source-level optimizations or transformations are integrated in established workflows.

4.2.5.2 Qt-Based GUI Mode

The text-mode user interface presented in the previous section is quite useful in automated setups. Since plain-text terminals have quite limited presentation capabilities, it is best suitable for setups where the actual optimization result is at focus. The second user-interface implementation provided within this framework focuses at a user-convenient presentation of processing step's operation. This can be best achieved in a graphical environment. To save effort and provide an established API, the well-known Qt user-interface framework has been chosen for this user-interface abstraction.

The graphical user interface implements the user-interface abstraction services as described in Section 4.2.4. Therefore, it can be used as a drop-in replacement for the standard text-mode user interface. According to the modular concept of the MACCv2 framework, the user-interface implementation can be encapsulated in dedicated libraries. Currently, the command-line executables presented in the previous section exploit this concept. They provide a command-line option to choose a particular user-interface implementation, enabling them to operate also in graphical environments.

Both, the logging and the progress indication service are realized in a common window. Figure 4.7 shows a GUI-based representation of the logging and progress indication. Basically, the upper part is covered by the logging facilities, while the bottom part

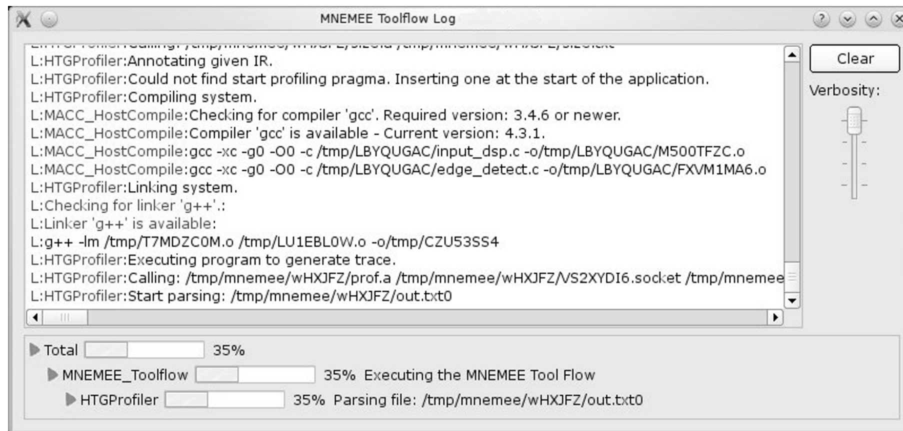


Figure 4.7: Logging and progress indication window.

contains the progress indication. Logging follows similar line-oriented representation as for the text-based output. The output of all four report types is presented in the sequence of its occurrence. For each type the caption text is highlighted in a different color. Further, the user has the possibility to control the output verbosity.

The progress indication section at the bottom of this window takes advantage of the graphical representation. In contrast to plain-text mode, where only the most recently changed progress is reported, here the full hierarchy of progress reports is presented to the user.

The input services of the user-interface abstraction are exposed to the user as dialogs. Figure 4.8 shows such an input dialog asking the user to select a particular processor in the system description.

In contrast to the text-mode user interface, the GUI provides two types of generic object views. The first is capable of presenting HTML pages to the user, the second shows PNG images. When requested by a processing-step implementation, each view instance opens a dedicated non-modal window showing the related object's content to the user. Subsequent update requests are also supported. Therefore, both object views are well-suitable for detailed presentation of intermediate results (e.g. an evolutionary algorithm may present its current population distribution in an image map).

Typically in the development process of project-driven toolflows, besides the pure processing-step implementation an adequate presentation and user interface is required for a successfully project outcome. The graphical user-interface implementation has been designed to provide a foundation for toolflow-dedicated user interfaces. This can be observed in a twofold internal structure. All graphical elements (i.e. logging facilities, object views, progress indication) have been designed as reusable widgets. In the standalone UI these widgets are located in dedicated windows, for a toolflow-specific user interface these widgets can be used either in the same way, or integrated in a toolflow-specific view.

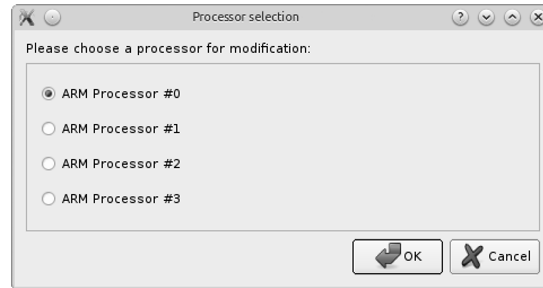


Figure 4.8: Selection input window.

Summarizing the capabilities of the graphical user interface, it can be concluded, that it provides a user-friendly drop-in replacement for the default text-mode user interface. Furthermore, it has been designed to be a starting point for dedicated user-interface development. In the context of the MNEMEE project such a dedicated toolflow interface has been developed with a significantly-reduced effort based on this foundation.

4.2.5.3 Eclipse-IDE Plugin

The third user-interface implementation targets the workflow in an integrated development environment. The user-interface abstraction has been designed to smoothly integrate into the Eclipse-IDE. In addition to provided user-interface abstraction services, focus has been put on interactive system-model design.

Since the Eclipse-IDE operates in a graphical environment, this UI implementation can be classified as highly interactive graphical UI abstraction. In general, the Eclipse-IDE provides dedicated spaces within the IDE View for similar services which match well the MACCv2 UI abstraction. Figure 4.9 shows a typical Eclipse workspace layout with the MACCv2 system-model editor opened. It is a drag-and-drop editor for system-model designs. Users may choose from a range of available system components and channels a particular one and drop it onto the drawing area representing the system model.

Within this area, modifications and removals of present components and channels are possible. Overall, fine-grained control over the system-model structure and properties of each component is provided. Furthermore, interconnections between components and channels and especially the address space mapping rules therein can be edited as well.

A complete system description consists of a structural system model and associated application code. Such associations can be created, modified and explored within the Eclipse-IDE. Each compilation unit (aka. source-code file) can be viewed and edited the same way as done for plain source-code projects within the IDE.

System-model descriptions are handled in a project-oriented way. Due to the framework design, system descriptions are represented in a self-contained file. Therefore, loading or storing of system descriptions affects the system model including the associated source code.

Once a particular system-model instance has been prepared, typically individual optimization and transformation techniques or complete toolflows are going to be applied

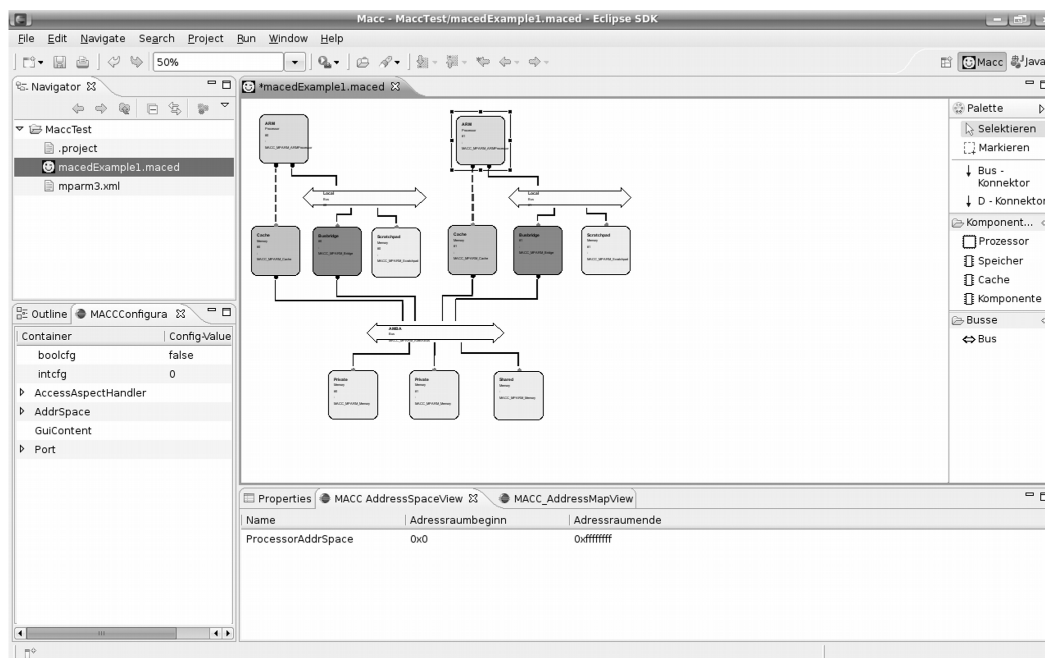


Figure 4.9: Eclipse user interface.

to it. Tools are typically accessible via the context menu in the system model. Since processing steps may have different scopes (i.e. some are defined to be applicable to processing units some at system level) the context menu takes this into account. Depending on which component has been selected, the applicable set of tools is displayed to the user.

General user input requests triggered from particular processing steps are handled in a similar way as for the Qt GUI. They are exposed to the user in dedicated dialog windows. Finally, no dedicated object views are provided within this UI currently.

In contrast to the text-mode UI and standalone GUI implementation, where abstraction of user-interface services for optimization and transformation techniques was at focus, the Eclipse-IDE integration focuses more closely on interactive design of system models. The user has a better fine-grained control over the structure of the system model, enabling manual tweaking of system properties to perform interactive design space explorations. Nevertheless, the required services to invoke optimization techniques or other processing steps are also provided. Therefore, the Eclipse-IDE-based user interface is best suitable for users who want to become familiar with the MACCv2 base optimization technique development process as well as those users in the early stage of an optimization technique development when the problem space is going to be explored and eventually a target platform has to be chosen.

4.3 System Modeling

A common system model is a vital prerequisite for well-cooperating optimization techniques within a multi-step optimization toolchain. Especially, in the optimization context targeted in this thesis, knowledge of target-platform properties is necessary to each memory-aware optimization step. Since optimization techniques take decisions based on the target-platform properties (i.e. memory sizes or latencies), inconsistent platform models may lead to contradicting optimization decisions which diminish the overall optimization result. Therefore, one of the major contributions of this thesis is the proposal of a unified target-system model which is accessible in each optimization and analysis step. The conceptual details have been presented in Chapter 3.

A system-model definition is encapsulated by a top-level object, derived from class `MACC_System`. Following the structure shown in Figure 4.10, the system-level object keeps track of the major building blocks of this system description. References to the set of components and to the set of channels are organized in separate container-based collections. Refer to Section 4.2.1.9 for more details on the container API. Following the system-model hierarchy, components are presented next.

A component has to keep track of the set of address spaces associated with it. Further, to represent interconnection relations, a component exposes ports. References to them are also retained within the component class. Finally, components participate in the computation of system-wide properties. The contribution of each component is represented via associated aspect handlers. Based on a generic component representation, a set of more specialized component types is provided. In particular there are predefined component types for processing units, memories, scratchpad memories, bus bridges and caches. To support annotation of properties to component groups, a representation of component classes is provided as well. A component class may be used to represent object relations to a set of components. For example, associating application code not to a dedicated component, but to the whole set of components which are of a particular type can be done by associating it to the class representation for this particular type.

Channels expose a similar structure. A set of address spaces is included. Further, a set of aspect handlers for modeling a channel's contribution to system-wide properties can be defined. Finally, references to ports connecting to this channel are preserved. Specializations of channel classes are provided for bus-based channels and direct-link channels.

Port representation is located at the next deeper hierarchy level. Ports connect components and channels together. Therefore, ports need to keep references to both items. Since, according to the system model, components expose ports, the port-component relation is fixed, while the port-channel relation can be modified according to the system-model structure. Once a link between a component and a channel has been established,

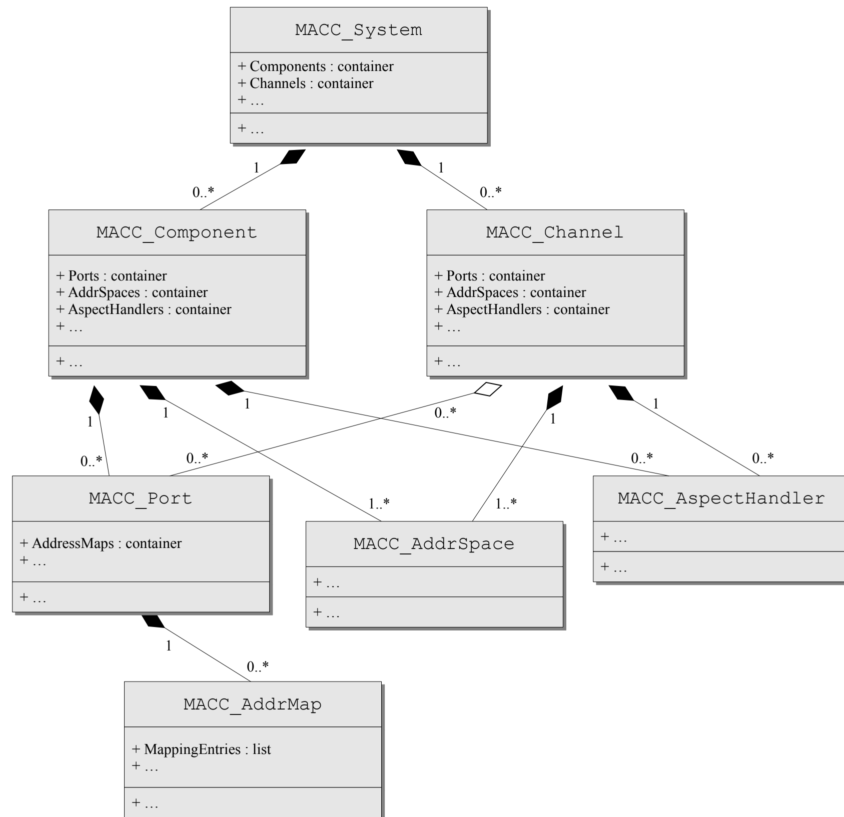


Figure 4.10: System-modeling API.

semantic properties of this link have to be described. In terms of this system-modeling approach, focus is directed towards translation rules which map address values in the context of a component to corresponding values in the channel's context, and vice versa. Following the unified structure, such mapping rules are defined within a dedicated container.

4.3.1 Practical Component and Channel Models

Within the following section, the set of system-model components and channels provided to the user is described. General classification according to entity type represented in the target system is performed. The focus is put on an overview of available components and channels as well as on their aspect handlers.

In advance, an overview of system-model templates is presented. These templates can be used to instantiate a full-system description according to the structure of corresponding target system. Within the MACCv2 framework, a system model for the MPARM simulator [59] and an abstract model of the host platform are provided. The MNEMEE project contributed further system models, namely, an ARM-based CoMET

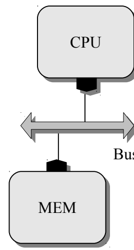


Figure 4.11: Host system-model template.

simulator [62] description and a Freescale MSC8144 [63] system model. These four system models serve different purposes. This primarily affects the level of detail and the implementation of aspect handlers.

4.3.1.1 System-Model Templates

Each system-model library provides component and channel classes which can be instantiated on request. The graphical frontend uses this feature to provide the on-screen toolbox of components and channels. Besides this individual instantiation method, a system model provides a full-system template. This template instantiates the set of components and channels according to the setup found in the target platform. These components and channels are connected appropriately. Furthermore, address mappings, aspect handlers and tags are initialized. Using these templates simplifies the system-model construction process. Especially in automatic toolflows where the tools are expected to operate on one or few predefined platforms, the system templates can be used to provide a simplified user interface.

Host System Model: The host system platform defines an abstract minimalistic platform which is intended to provide a generic host system representation. The primary purpose of this platform is to encapsulate the host system's compilation and linking tools into the same interface as for actual target-platform tools. The correct representation of all details of a particular host system is not at focus. Therefore, the system structure implemented here consist only of necessary components: The processor core, for the application-code representation and a memory component for representation of storage space and data or code assignment. Both are connected via a generic bus. Figure 4.11 depicts a host system as created according to the system-model template. This system template has no configurable options.

MPARM System Model: The MPARM system-model template defines a system model structured according to default MPARM setup. A configurable number of ARM

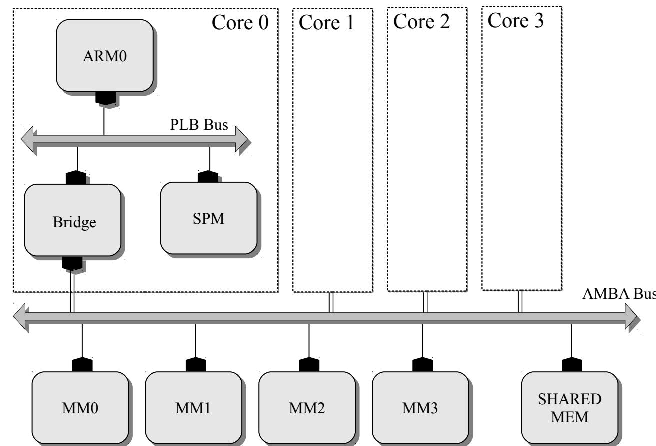


Figure 4.12: MPARM system-model template.

core components is instantiated. Each core has an associated scratchpad-memory component. The size of this memory is configurable as well. Furthermore, a private main-memory component of configurable size is created. Finally, a single shared-memory component is created. These components plus additional bridging components are connected via buses. Two types of bus channels exist. The first one represents the processor local bus. An instance of this channel is created for each processing unit. The scratchpad-memory component is connected to this bus. The second type represents the system-wide AMBA bus. According to the MPARM model, private main-memory components and the shared-memory component are connected to this bus.

Figure 4.12 depicts the MPARM system model as created according to the template with default parameters. The parameter values match the configuration parameter of the MPARM simulator version as used in MNEMEE project:

- Number of Cores: 4
- Scratchpad: 12 kB
- Private main memory: 12 MB
- Shared memory: 16 MB
- Annotated processor core frequency: 200 MHz

CoMET ARM System Model: The CoMET ARM system-model template constructs a system description similar to the MPARM system. The development of the CoMET system description was primarily motivated by the need for a drop-in replacement for the MPARM environment within the MNEMEE project.

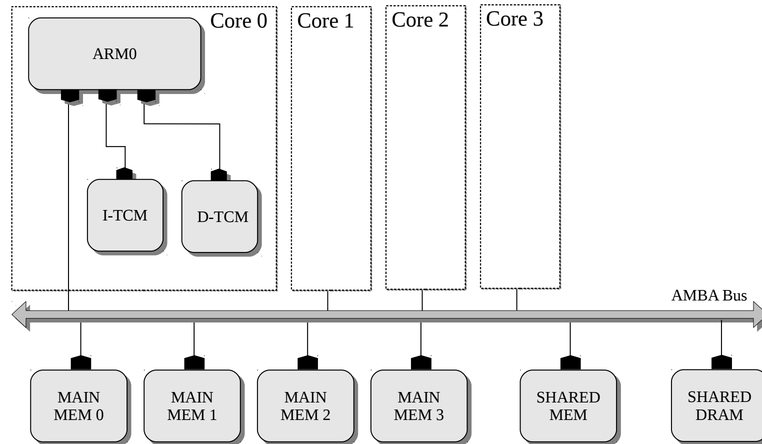


Figure 4.13: CoMET system-model template.

The overall structure is similar. The system-description template constructs a configurable number of cores. Per-core memories and a single shared-memory component follow. In contrast to the MPARM system model, tightly coupled memory components are not connected via a processor local bus, but direct links and additional ports on the core are defined. Furthermore, an additional secondary shared memory is provided.

The set of configuration parameters is similar to the MPARM system template. The system description depicted in Figure 4.13 has been constructed according to following default setup values:

- Number of Cores: 4
- TCM: 16 kB instruction TCM, 16 kB data TCM
- Private main memory: 16 MB
- Shared memory: 1 MB
- Shared DRAM: 512 MB
- Annotated processor core frequency: 500 MHz

MSC8144 System Model: The MSC8144 system-model template constructs a system model of corresponding Freescale MPSoCs. For the targeted usage scenario within the MNEMEE framework an abstract system model has turned out to be sufficient. The focus has been put on memory-allocation strategies for the StarCore processing units. Therefore, the abstract system model contains only the StarCore processing units and the corresponding memory subsystem. To be able to represent the memory partitioning and provide adjustable energy and latency values for each partition at minimal effort, a diverging implementation of on-chip hardware structure of the memory subsystem has

been chosen. This way, predefined aspect handlers could be reused. The MSC8144 hardware connects all memories via a system-wide CLASS crossbar. In contrast to this structure, the system model employs abstract buses to group the memory partitions and corresponding processing units. The resulting overall system-model structure shows the expected hierarchical structure which would have been otherwise obscured, if the model had exactly followed the actual hardware structure.

The general structure is: Four processing-unit components, each has access to private memories at each hierarchy level plus access to one shared memory per hierarchy level. There are three hierarchy levels in the memory subsystem.

The system-model template has various configuration parameters. The first set of parameters can be used to adjust the sizes of private memories at each hierarchy level. A rich set of parameters provides the configuration input to the energy-consumption and latency computation aspect handles. Individual parameters for memories of each hierarchy level can be defined.

A typical system description of the MSC8144 platform is depicted in Figure 4.14. Following default configuration values have been used:

- Private L2 memory: 64 kB, Latency: 1 cycle, Energy consumption: 1 pJ
- Private L3 memory: 512 kB, Latency: 10 cycles, Energy consumption: 10 pJ
- Private DDR memory: 16 MB, Latency: 100 cycles, Energy consumption: 100 pJ
- Shared L2, L3 and DDR memories: Same as private memories
- StarCore: Energy consumption: Idle: 1 pJ, Active: 2 pJ
- StarCore access setup overhead: 1 cycle

4.3.1.2 Processing Units

Processing-unit components described in this system-modeling approach have a common base class:

This processor class implements an interface for association of application code to a particular processing-unit component. The semantics of such an association relates an application code as being intended to be executed on that particular processing unit.

Host Processing Unit: The host processing-unit component, as part of the host system model, does not target a particular physical processing unit. The purpose of the host model is to provide an abstract environment in case no particular target platform is selected yet. The host processing-unit component does not expose a lot of specialized properties. Primarily, the inherited capability to associate application code has

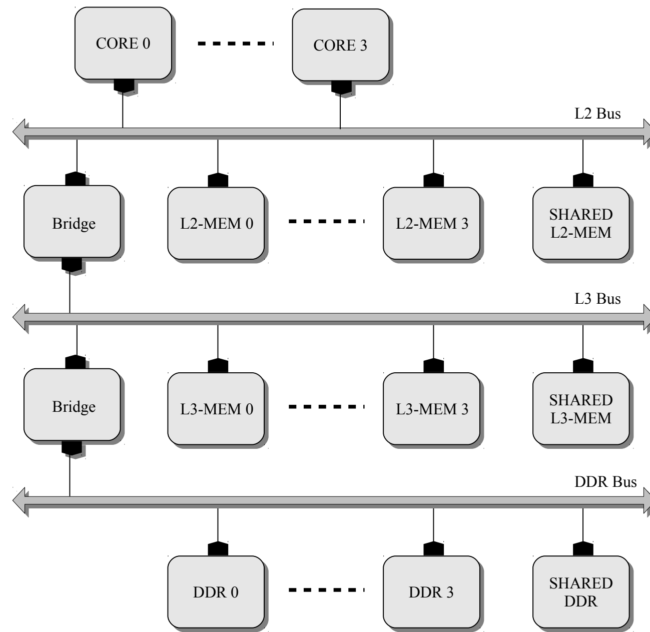


Figure 4.14: MSC8144 system-model template.

been combined with dedicated compilation and linking tools fitting the host compilation toolchain. Integrating seamlessly the host compilation tools into the framework enables fast development of profiling and analysis approaches.

`MACC_HostProcessor`

MPARM Processing Unit: This processor model targets the SWARM core included in the standard MPARM distribution. This is an ARM7-based unit. According to the modeling approach, a dedicated component class is provided, uniquely denoting this processing-unit type:

`MACC_MPARM_ARMProcessor`

The base class is the generic processor class. According to the ARM architecture, this processor component operates on a single address space. Data and application code are located within that address space. The address space covers a 32 bit address range. Since the processor provides a single initiator port, the entire range of that address space is mapped for any kind of access to that port. This resembles the modeled ARM architecture, where the core interfaces only to a local bus, called PLB.

The MPARM processing-unit class provides predefined aspect value handlers for energy-consumption computation as well as for latency computation. Both handlers resemble the energy and timing model used in the MPARM target.

Not immediately related to the system-modeling approach, nevertheless dedicated to the MPARM processor component, are MACCv2 tool specializations. The MPARM uses target-specific compilers and linkers. Therefore, the corresponding MACCv2 tools need to be adapted for that platform. The MPARM system-model library provides compiler and linker tools which invoke the MPARM GCC-based toolchain.

Besides these properties, a set of tags provides optimization techniques with hints on the processor type. A tag identifies it as a `RISC` processing unit. Another exemplary tag provides information about the clock `FREQUENCY`.

CoMET Processing Unit: The processing-unit component for the CoMET platform targets an ARM11 processing core as provided within the CoMET simulator. The component class is denoted as:

```
MACC_COMET_ARM_ARMProcessor
```

It is derived from the generic processor class. Since representing an ARM core, a single address space is defined. In contrast to the MPARM system model where the core performs all accesses via a single interface, the ARM11 core model embeds tightly coupled memories. To reflects this structure, the CoMET processor component has dedicated ports which connect via direct links to such local memories.

Further properties are similar to the MPARM core. Aspect handlers for energy and latency computation are provided. Tags indicate further processor properties.

MSC8144 Processing Unit: The MSC8144 processing-unit component models one of the four StarCore DSPs present in the target platform. Similar to the MSC8144 system model, the processing unit does not target a detailed representation of the actual StarCore. Therefore, the processing unit is kept at an abstract level. The structure exposes a single address space and a single port to the memory subsystem. Similar to the other processing-unit models, this one provides aspect handlers for energy-consumption and latency computation. Since no actual energy and latency model for these cores was available, the general computation scheme is based on the MPARM aspect handlers. In contrast to them, several configuration parameters have been added enabling adjustment of fundamental values, once measured data becomes available.

```
MACC_MSC8144_Processor
```

4.3.1.3 Memories

The set of memory components is presented according to available system models. The base class for any kind of memory component is named

`MACC_Memory`

To enable simplified memory-access optimization techniques, which need only to distinguish between fast local memories and some main memory, another subclass has been provided:

`MACC_Scratchpad`

This class indicates such a local memory. Any kind of higher-level memory is going to be represented by a class derived directly from the generic memory class.

Host-system memory: The host-system memory represents a hypothetical main memory. Since the host system model targets an abstract intermediate platform, the memory component follows the common assumption of a huge or infinite main memory. Therefore, the entire 32 bit address space range of the host-processor component is covered by such a memory component. No default aspect handlers are provided for the host-memory component.

`MACC_HostMemory`

MPARM memory: The MPARM system has two types of memories:

`MACC_MPARM_Memory` `MACC_MPARM_Scratchpad`

`MACC_MPARM_Memory` represents private and the shared memories. The second type, the scratchpad memory `MACC_MPARM_Scratchpad`, represents the tightly coupled processor local memories.

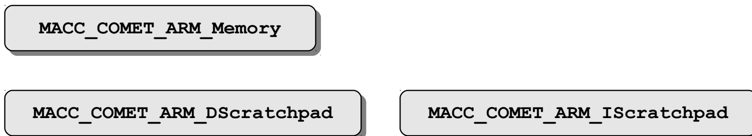
Each memory component has an address space with a configurable range. This enables the reuse of the same component class for private and shared memories in the system model.

The MPARM memories provide aspect handlers for energy-consumption and latency computation. The energy-consumption model follows the scheme applied within the MPARM. Three access sizes of 8, 16 and 32 bits are distinguished. In the orthogonal dimension a distinction between read and write accesses is performed. This six value matrix is provided at construction time of the memory component to characterize its energy-consumption behavior. Recent extension performed in the context of the MNEMEE project by Jovanovic [64] adapts this set of values according to the address space size of a particular memory component. The adjustment is based on factors retrieved via CACTI [65].

The latency computation assumes a fixed per-access value regardless of the access type. Theses values are configured on instantiation of memory components.

The tightly coupled memories (aka. Scratchpads) in the MPARM model expose the same energy-consumption and latency computation models as the main memory components. Therefore, the aspect handlers for both value types could be reused. Different initialization parameters reflect the significantly lower per-access energy consumption and latency.

CoMET memory: Within the CoMET system model, two basic types of memories are provided. The first one identifies shared and local private memories. The second one targets the tightly coupled memories (TCM) associated with each processing unit. In contrast to the MPARM system model, a further distinction between data TCM and instruction TCM is done. Therefore, the CoMET system model defines three classes of memory components:

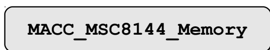


The memory component class is used for modeling system-level memories. The scratchpad components are used for representation of tightly coupled memories.

Energy-consumption estimation and latency computation are related to the MPARM model. Therefore, all types of memory components utilize the same set of aspect handlers. Energy-consumption aspect handlers require similar six value matrix for initialization to determine the per-access energy consumption for 8, 16 and 32 bit accesses.

Latency computation requires a single value. The aspect handler accumulates these latency values for all types of accesses. This uniform behavior resembles well static RAM properties.

MSC8144 memory: The MSC8144 system model defines one class of target-specific memory component:



This is the common class to all memories at all three hierarchy levels modeled in this system description.

Energy and latency aspect handlers are provided for MSC8144 memories as well. For the purposes of the MNEMEE project a simplified energy and latency model was sufficient. Within this model the memory has a fixed energy consumption and a fixed delay per access, regardless of the access size and type.

4.3.1.4 Caches

Similar to memories, a common base class exists for caches. From the conceptual point of view, cache components can be regarded as bridging elements with scenario dependent forwarding of accesses to the next level. This matches well with the behavior of actual

caches. Assuming the best-case scenario, the requested data is present in the cache, resulting in a cache-hit. Since the data does not have to be fetched from the next level, the best-case scenario does not consider sub-accesses below a cache component. Opposed to this, the worst-case scenario assumes a cache-miss, which requires the data be fetched from the next level. Therefore, the subtree needs to be considered in aspect handler computations.

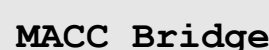
Since the system-modeling approach, and in particular, the aspect handler implementation, is considered to be stateless across multiple access queries, the cache aspect handles have to rely on the scenario selection provided by the user. Nevertheless, the possibility to place access requests consisting of an access sequence is especially beneficial for aspect handling at caches or similar components. Within such an access sequence an order of accesses exists. This can be used to refine the aspect computation. For example, despite a worst-case scenario an aspect handler may still assume cache-hits if subsequent accesses to the same cache line are recognized.

4.3.1.5 Other Components

Processing units and memories are the components at focus of this system-modeling approach. Nevertheless, other component types may also be required for a sufficiently precise target-system representation.

Most commonly used components within this class are bridging components. They link together two communication channels at different hierarchy levels. Since acting transparently and usually requiring only limited hardware effort in physical systems, they are often omitted in common structural system diagrams. Nevertheless, for a well-structured system model, no two communication channels may be connected directly to each other, and therefore within this system-modeling approach such bridging components are required once an interconnection hierarchy is going to be modeled.

Bridges: A bridge component exposes two ports: one target port and one initiator port. The target port receives requests to be placed on the subsequent channel via the initiator port, eventually with remapped addresses. Except for the host system model the other three system models presented in this section incorporate bridge components. The components can be identified according to their object class. The framework provides a common base class called:



DMA Units: The second type of additional system-model components are DMA units. With respect to the access-based view within this system-modeling approach, DMA units are similar to processing units. Both initiate accesses to the memory subsystem. Due to the limited computational power, usually no application code can be associated directly with a DMA unit. Nevertheless, defining such components and incorporating them into the system model has several advantages. Especially, if the application code is going

to exploit data movement via such unit. The optimization technique introducing such DMA-based copy operations may use access-based requests to determine the correct setup (i.e. translate addresses between the processing-unit view and the DMA view of the memory subsystem). Furthermore, the same aspect handler-based system properties can be requested relative to DMA units, effectively providing a uniform cost model for processing unit and DMA-based accesses.

I/O Devices: Finally, a system model may contain components which are neither related to application code nor to the memory subsystem. Common examples are I/O components. Nevertheless, even in the case where primarily targeting memory-access optimizations, it may be advantageous to incorporate such components into the system model. The presence of such components increases the precision of a system model, which may improve the precision of computed aspect values. A real-life example is the AMBA bus energy model in the MPARM system. Its computed per-access energy-consumption value is dependent on the number of ports connected to this bus.

4.3.1.6 Buses

To connect system components together and form a complete system model, a representation of communication channels is provided within this system-modeling approach. The base class for any communication channel is defined as:

MACC_Channel

Based on this, a distinction between point-to-point links and shared access channels is performed. Currently, for the shared access channels one type of common base class exists:

MACC_Bus

As the name indicates this is the base representation of any kind of bus-based communication channels. This is the most common communication channel type. All four system models presented in this section provide derived bus implementations.

Host bus: The abstract host system model consists basically of one processing unit and one memory component. For their interaction a bus channel is needed. Due to the abstract nature of this system model, the communication channel does not provide any specialized features. A subsequent class defines the unique type of this bus. No default aspect handlers are provided:

MACC_HostBus

MPARM busses: The MPARM system model defines two buses:

`MACC_MPARM_PLBus` `MACC_MPARM_AMBABus`

The PLBus channel class defines the processor local bus. This bus primarily connects a processor core to the local scratchpad memory. Within the MPARM simulator the PLB does not contribute to the computed energy consumption and does not expose additional latencies. Therefore, the corresponding system-model channel has fairly simple aspect handler implementations. Both default aspect handlers for energy-consumption and latency computation are framework-provided standard handlers preinitialized with zero contribution.

The AMBA bus is provided for the second hierarchy level. The MPARM simulator computes energy-consumption and latency values for this bus. Within the MPARM system model both aspect handlers are also provided. The per-access bus latency exposed within the MPARM simulator is not a constant value. The AMBA bus energy-consumption estimation requires more effort. MPARM computes the energy consumption in an iterative approach. A per-cycle value is computed according to the signal setup for each cycle. These values are accumulated to a per-access total energy-consumption value. The AMBA bus component within the MPARM system model utilizes the same computation method. In contrast to the MPARM simulator, the energy aspect handler does not keep signal state across multiple accesses. This leads to a slight imprecision compared to the actual MPARM values. As can be observed in the results chapter, the deviation has only a limited effect, with the overall per-access energy value deviation being stable and sufficiently low.

CoMET busses: The CoMET system model differs in the way tightly coupled memories are bound to the processor. In contrast to a MPARM system, these memories are connected to the processor component via dedicated links. Therefore, the PLB is not modeled here. The processor cores connect immediately to the AMBA bus.

The CoMET AMBA bus model is based on the MPARM AMBA bus model. The aspect-handler implementation follows the MPARM model. A fixed bus latency per access is contributed. Energy computation is performed in the same way as for the MPARM model. Since the CoMET platform does not define an energy-consumption model, the system description uses the MPARM values for a rough estimation. The class name for this bus component is denoted as:

`MACC_COMET_AMBABus`

MSC8144 bus: The MSC8144 system description defines one bus component:

`MACC_MSC8144_Bus`

The MSC8144 system template instantiates three buses of this class. These buses are connected via bridging components to form a hierarchical system model. The decision to use the same bus implementation for each hierarchy level is feasible for the high

abstraction level chosen for this system description. In line with this abstraction level, the default configuration assumes these buses to be overhead free, both, in terms of energy and latency.

4.3.1.7 Direct Links

Practical implementation of direct-link channels are provided within the CoMET system description. Within this system description, direct links are used to connect tightly coupled memories to processor cores. The CoMET-specific direct-link class is a direct descendant of the generic direct-link channel class. Since these links are considered to be overhead free within the CoMET platform, the aspect-handler implementation uses standard fixed value handlers preinitialized to zero overhead. The corresponding class implementation is denoted as:

```
MACC_COMET_ARM_Directlink
```

4.3.2 Common Aspect-Handler Examples

This section demonstrates the aspect-handler implementation on several examples. These examples target both a real-life example as implemented in the MPARM system model and some further, less common examples which show the flexibility of this approach. The first example describes the aspect values provided within the MPARM system model. Energy-consumption and latency values are provided. Compared to the actual MPARM platform the values computed here are quite precise. A deviation of less than 0,21% has been achieved. Refer to Section 5 for a more in-depth analysis of these results. The MPARM system model provides specialized aspect handlers for energy-consumption computation and latency computation. Both aspect-handler types resemble the actual energy and timing estimation provided within the MPARM simulator.

The MPARM model consists of following specialized components:

- ARM7 Processor component
- Scratchpad memory component
- Configurable common memory component for private and shared memories
- Bus bridging components

Furthermore, two specialized channel types are defined:

- PLB - 1st level Processor local bus
- AMBA Bus - system-wide bus connecting the processing cores to the shared resources

Since the focus has been put on precise estimation of energy-consumption and latency values for memory accesses, I/O components have been omitted in this system model.

Latency Handlers

Latency computation is performed for aspect values identified under the default name `Aspect-Cycles`. For the aspect-value representation, a MACCv2 framework-provided data type is used. This consists of a floating point number plus a unit representation. Such a combined representation is beneficial once operations on differently scaled values are going to be performed. A combined data type implementation ensures correct scaling while performing arithmetic operations. The unit for latency values has been defined as "Cycles". Since the MPARM does not provide sophisticated control over the clock rates applied to each component, the latency cycles are always related to the global system clock.

Components within the MPARM simulator expose a constant timing behavior. Therefore, for most components in the system model, plain accumulating aspects handlers with preinitialized increment values can be used. The single component which requires a specialized handler is the bridge. Within the MPARM, the bridging between the PLB and AMBA Bus introduces different delays depending on the type of access. In detail: a read access can be performed free of overhead, while a write access always introduces a delay of one cycle.

Energy Handlers

With respect to the energy model, specialized aspect handlers are required. Except for the PLBs, which are considered to be overhead free in terms of energy and latency, for all other component classes dedicated aspect handles exist. The energy-consumption model provided for the MPARM system description consists of a common energy-consumption model and corresponding aspect handlers for all kinds of memories, an energy-consumption model for each ARM processor core and the model of the AMBA interconnect.

Energy-consumption estimation is performed for aspect values named `Aspect-Energy`. Similar to the latency values, the energy consumption is represented as a floating point number and the corresponding unit. The energy consumption is measured in Joule, therefore, the unit is defined as "J". Typical magnitude is in the order of 10^{-9}.

The energy model and the corresponding aspect handler for memory components distinguish between three data sizes and whether it is a read or write access. In particular distinct values for 8 bit, 16 bit and 32 bit accesses are defined. The aspect handler for energy-consumption estimation chooses according to the access request size the corresponding value. The MPARM simulator applies for any kind of memory the same energy-consumption model. The same way MPARM system description has been implemented.

The second type of energy-consumption estimation aspect handler computes the contribution of processing units. The energy consumption of a processor is abstracted according

a two state model. According to this model, performing an access requires a number of cycles in an active state plus further cycles in an idle state while waiting for the memory subsystem to provide the data. In contrast to the memory aspect handler the processor energy consumption depends on accumulated latency of the sub-tree. Since the value computation is performed bottom up, the energy aspect handler at a processing unit has access to this value while being invoked.

Finally, the AMBA bus also contributes to the overall energy consumption of a MPARM system. A dedicated energy-consumption aspect handler is provided for this bus. The energy consumption of the AMBA bus is computed on a cycle-by-cycle base. The energy model defines per-cycle contribution according to the states of bus signals within that cycle. Similar to the processing-unit-related handler, this one needs the sub-access latency to introduce sufficient waiting cycles. In contrast to the MPARM bus energy model, this one does not keep an access history across multiple requests. Referring to the results section, this is the source of minor imprecision of this system-model description compared to the actual MPARM full-system simulation.

Both energy and latency values of sub-accesses contribute in an additive way to the overall values at each access tree node. Scenario selection is implemented as the choice between the maximum (in worst-case scenario) and minimum (in best-case scenario) combined value.

The MPARM energy and latency aspect handlers served as the foundation for the motivating example in Section 3.4.2 demonstrating the value computation. Therefore, the computational steps shown there apply also to the MPARM aspect handlers.

Exemplary String Concatenation Handler

The following example shows an exemplary string-based aspect-value representation. The aspect value is basically an unstructured sequence of characters. The aspect handler performs common transformations on such a character sequence. Appending, inserting and cutting subsequences are the well-known ones. A practical example of string-based aspect values would be the construction of user-friendly names for corresponding energy or latency values. Basically, an aspect handler implementation would set the string to the component name, add some separation mark (i.e. "→") and append to this the string of the subtree selected according to the scenario. A MPARM system description equipped with such aspect handlers for the aspect value `Aspect-Path` would generate from an access targeting the shared memory following path string at the top-level access: `CPU→PLB→BRIDGE→BUS→SHMEM`

Even though this aspect value does not require very sophisticated computations, it is quite useful. Generating such name strings for accesses is valuable either for debugging purposes or for obtaining unique identifiers for data computed in such accesses.

Exemplary Data Storage Handler

Finally, another example of versatile aspect handler application are aspect handlers which implement represented storage space of memory components. They need to be accompanied by aspect handlers which perform a simulation of corresponding data value transport. This applies to buses or bridging components. The implementation basically copies the data according to the access specification from or to the sub-access. Scenario selection will be applied to choose the unique source location for each value. The aspect-value representation in target component handlers would be an array of data values. The size of such a data array needs to be equal to the access request size. The action to perform on this data is defined according to the access specification (i.e. read or write). Since in general an access request may target several locations due to the chosen size and range parameters, the aspect-value representation needs to provide an additional symbol for undefined or ambiguous values.

In combination with latency and energy handlers the data value handler provides a foundation for a simple memory-subsystem simulator with energy-consumption and access-time estimation.

According to these examples, the aspect-handler model has a wide application scenario. Especially, the unrestricted value representation combined with a C++-native aspect-handler implementation provides a flexible approach to derive system-wide properties. The access-based perspective extends common iterative system-model exploration approaches towards an often demanded processing-unit-centric perspective.

4.4 Processing Step Integration

A framework which targets optimization technique development needs to support common optimization implementation approaches. Mostly due to the complexity and often also due to intended reuse, optimization techniques are implemented as a sequence of processing steps. This section presents implementation details for such a processing-step integration according to the MACCv2 framework.

The MACCv2 framework presented in this thesis provides an optimization technique integration service which matches the requirements of a state-of-the-art optimization development process. As sketched in this section, support for subdivision of complex optimization techniques into a set of processing steps, denoted as “tools“, is the key integration concept provided within this framework. Such tools expose unified APIs for enumeration, invocation and dependency tracking. Since tools may invoke sub-tools, the required support for hierarchical processing-step implementation is also present within this framework.

4.4.1 Tool Implementation

Optimization or analysis-step implementation based on the MACCv2 framework are instances of classes derived from `MACC_Tool`. Applying these tools to system-model

objects leads to invocation of corresponding methods within a tool implementation. To achieve a uniform invocation scheme, the `MACC_Tool` base class declares the common interface methods, which need to be implemented in each tool. Basically, these are a few methods describing properties of a tool. Examples are flags defining whether the tool may be applied several times to a particular object or in the case the tool performs modifications to the object, whether these modifications are reversible. A second set of methods implements the invocation API. Currently, three actions are defined:

- Setup a runtime environment for the tool (i.e. preinitialize environment variables etc.)
- Perform the actual tool action: in terms of this framework, "apply" a tool to one or several system-model items.
- Revert the changes performed by a tool. This action is denoted as "cleanup". Not every modification is reversible. Therefore, the tool may indicate to the framework that this action is not available.

Implementing processing steps as tools encapsulated in C++ object classes enables reuse of common-base-class-related framework services. Especially, two services are of importance: The first is the configuration API to steer tool operation. The second is the reflection API. Identifying tool class relations is a valuable prerequisite for the framework to integrate a tool specialization approach as presented in the next subsection.

From the developer's perspective, implementing optimizations and analyses as MACCv2 tools is reduced to a few mandatory steps, others are optional. This way a lightweight implementation can be achieved for the most common application scenarios.

In detail the mandatory implementation parts are:

- Implement the C++ object class derived from the `MACC_Tool` or some more specific tool class.
- Implement an initialization method, where at least the name, a description and the set of classes this tool can be applied to, are defined.
- Implement the action method which will be invoked whenever this tool is going to be applied to a suitable object.

Some of the most common optional parts to be implemented:

- Define dependencies to other tools which should be applied in advance to this one.
- Define additional classification tags.
- If the tool needs additional structured data as input or produces such data as output, define an interface class, which can be attached to system description or to application-code-representing objects.

- If possible, implement methods for reverting actions performed by this tool.
- Implement tool-settings creation and initialization methods.
- Implement tool-environment creation and initialization methods.

Once a tool is going to be used, the framework ensures appropriate object construction. To reduce instantiation overhead, the framework keeps track of currently available tools. In general, a single instance per tool is created at runtime and preserved until the process terminates.

4.4.2 Tool Specialization and Abstract Tools

The object-oriented representation of processing steps and class inheritance are used to implement a hierarchical representation of processing steps. Figure 4.15 illustrates the hierarchical relations for the class of compilation tools, starting at the root, which is always represented as the most generic class `MACC_Tool`. At this level only the basic invocation and configuration APIs are provided. At the next level all kinds of compilation tools are grouped. Typically, compilers are invoked as external executables. Therefore, another level of classification is added. Further specialization with respect to the invocation pattern and available options focuses on the often used GCC (GNU Compiler Collection) [66] as compiler framework. Based on this common class, platform-specific compiler tools are provided.

As can be observed in the case of compilation tools, a class-based approach is not only beneficial for identification of processing steps, but can also be used for reduction of implementation overhead for future processing-step types. In the case of compilation tools, the invocation of external tools and further common tasks can be implemented in a base class, reducing the effort for platform-specific tool implementations.

Besides overhead reduction, good practice software design approaches use class hierarchies to implement common interfaces. This approach is also applicable to the tool model provided within the MACCv2 framework. Since framework tools are based on the common-object-class model. The interface definition includes configuration API entries, enabling unified configuration entries for each class of tools. Typically, interface defining classes omit the implementation. In the case of processing-step tools, such interface tools would not be operational. The framework provides means to express this case. A distinction between regular tools and abstract tools is provided.

According to the class hierarchy, identification of processing-step types and corresponding hierarchical dependencies is possible. Based on this foundation, the framework provides methods to automatically choose the most suitable tool for a particular system-model object. The relation between tool implementations and system-model items is achieved via an interrogate callback at system-model item level. In the process of applying tools to system-model items, each item may suggest a more specialized tool instead of the one been requested. This specialization approach is, for example, widely used for compilation and linking tools. Since these are highly platform-dependent steps, each

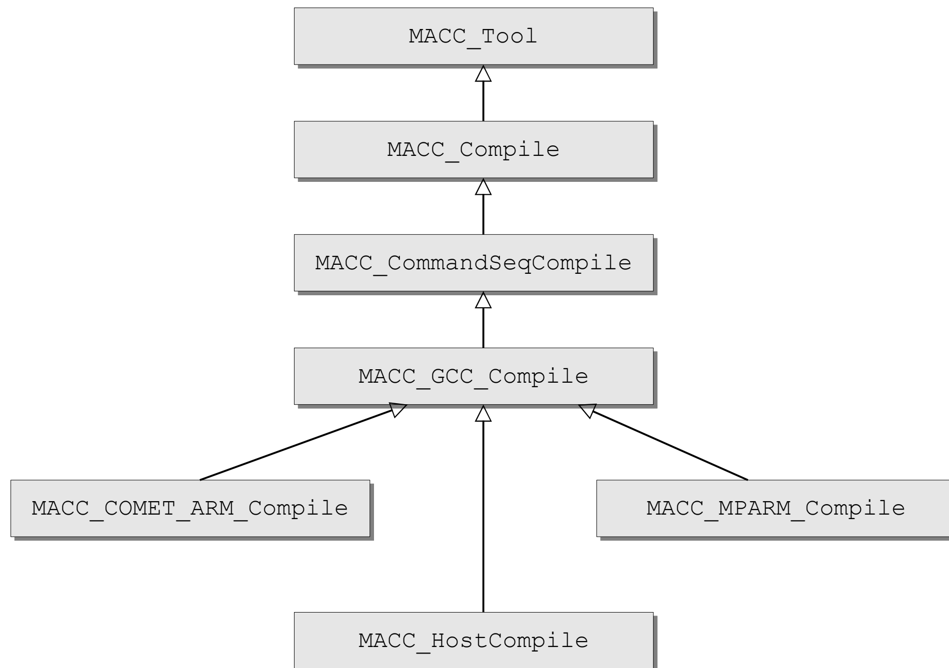


Figure 4.15: Compilation tool hierarchy example.

processing-unit component suggests a suitable specialized tool, which is used when compiling code for this processing unit. From the toolflow perspective, this simplifies the code generation process significantly, since regardless of the actual target platform, always the generic tool `MACC_Compile` can be requested to be used.

4.4.3 Processing-Step Interaction and Configuration Options

A typical processing step needs some input data and eventually produces some output. Since applied to particular system-model component, the system model and associated application code can be considered as implicit input data to each processing step. Nevertheless, processing-step-specific input data is likely needed as well. This kind of data can be annotated to relevant system-model components. The data is represented as so-called tool interface objects. Furthermore, most processing steps will need some means of controlling and configuring their processing actions.

A good practice recommendation for compilation and linking of MACCv2-based tools is to provide separate libraries for the actual tool implementation and the optional interface. On the one side, this approach opens the opportunity to reduce the memory footprint of processes which have to prepare input data, or have to deal with the results. They only need to use the interface library and do not need to pull-in the whole tool code. On the other side, this separation would also help, if licensing issues could prevent distribution

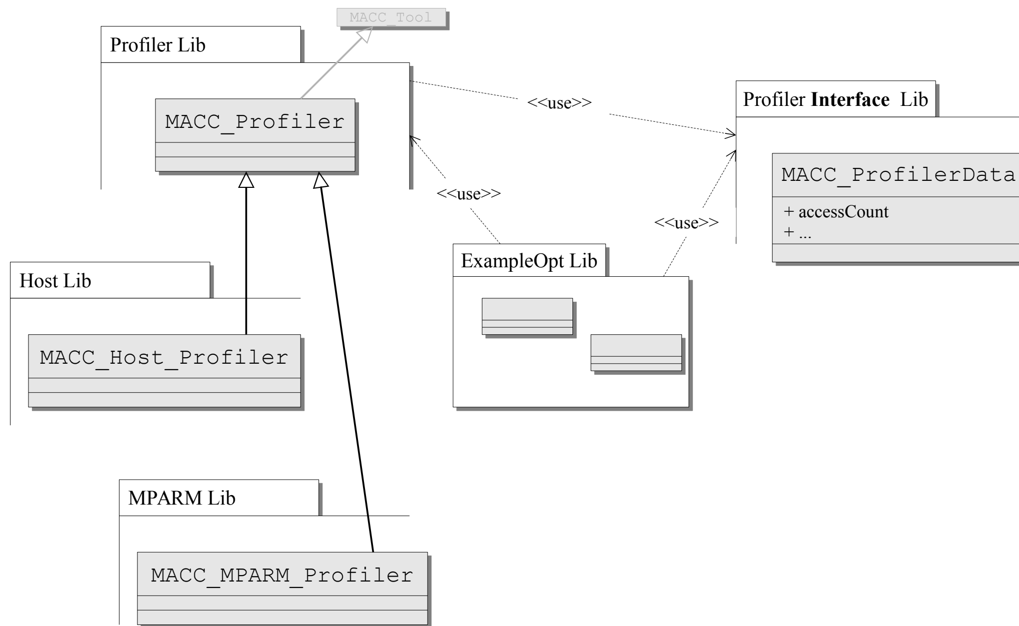


Figure 4.16: Tool interface example.

of particular processing steps even in binary form. In that case, processing could be performed locally and only the interface library would be distributed. Figure 4.16 depicts this good practice approach for such interface definition. The basic concept is to separate the optimization technique implementation and the interface definition. Each of them is located in a dedicated library. The interface library can be used in several tools, including both cases, tools which generate such interaction data, as well as tools which use it as input data. Especially, in combination with abstract tool definitions according to Section 4.4.2, the interface separation provides a powerful method to achieve true architecture-independent optimization-technique implementations. To summarize the effects, an architecture-independent optimization technique will typically consist of a set of sub-steps which are invoked according to their generic tool class. Since the interfaces are also defined at that level, these sub-steps can cooperate with each other regardless of the actual implementation being used for a particular architecture.

Continuing towards control over the actual operation of processing steps, a concept of tool setting and configuration options has been introduced. To demonstrate usage and implementation of these control instruments, a compilation tool example will be analyzed with respect to the available control options. In the straightforward case, a GCC-based compilation tool requires - among others - the configuration of the optimization level and a path to the actual gcc executable. The optimization level is a variable configuration option. Especially, in the course of the compilation process, the optimization level may vary depending on the processing unit the compilation tool is applied to. Various reasons

to vary the optimization level exist. Most commonly occurring is the expected usage of the compiled binaries. If the code is going to be used to run profiling steps, most likely a low optimization level is desired, which in turn will result in a straightforward mapping between the application code and the corespondent binary representation, while for generating the final executables typically a high optimization level is beneficial. The optimization level is defined best as a tool setting object.

According to the example in Figure 4.17, the compilation tool may request the framework to provide tool settings for selected system-model objects. From such settings instance the compilation tool can read the expected optimization level. Since, tool settings are not required to be present at each item of the system model, the framework tries to locate more generic tool settings according to the parent-child relations present in the system description. Nevertheless, a system description may contain no suitable tool settings, in this case the tool needs to define default values to stay operational. In the case of a compilation tool, this is a default optimization level. Such default options are expressed as configuration entries related to the tool. The tool developer has to choose basically between two places where to put such options: either immediately as entries of the tool object itself, or as entries of a default tool-settings object. Which one is more suitable, depends on the purpose of a configuration option.

Continuing the example, the other important option of a GCC compilation tool is the path to the gcc executable. Typically, the path stays unchanged over the runtime of an optimization technique. It is constant information, which is dependent on the operating-environment setup. Such configuration options are expressed best as MACCv2 environment variables. Such environment variables resemble the same well-know approaches in Unix-based operating systems. Actually, among various types of such environment variables, the framework provides means to link operating-system environment variables to corresponding ones within the framework representation, enabling configuration of framework tools from typical invocation script.

Concluding the configuration options, the processing-step model proposed in this thesis offers basically four ways to pass configuration parameters to a particular processing step:

- Tool-settings objects attached to particular components of the system model.
- Configuration fields in the default tool-settings object.
- Direct configuration fields in the tool object.
- MACCv2 environment variables.

4.4.4 Toolflow Construction

Based on the framework-provided integration services, tools can be combined into toolflows which perform intended optimization or analysis steps. In previous sections the most common aspects of such step-wise optimization technique development process have been presented. This section focuses on the actual construction of toolflows.

In the simplest case, the framework is capable of constructing automatic toolflows according to tool dependencies. In this case no explicit toolflow definition is required.

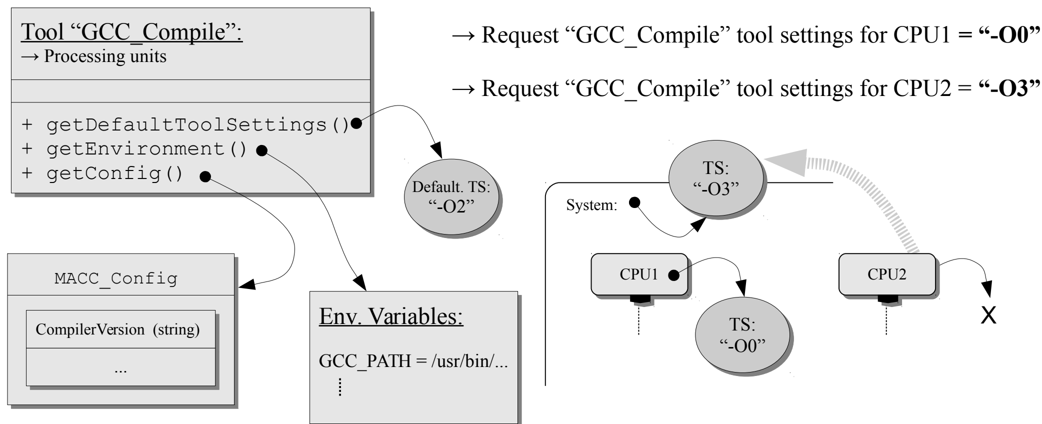


Figure 4.17: Tool configuration options.

Tools are invoked with their default configuration options according to their input requirements. The final tool will be the one specified by the user.

Since typically such simple cases occur only for subsets of processing steps, in fullfledged toolflows an instance coordinating the tool execution is required. Such an instance may execute several subtools conditionally, or eventually introduce loops in the tool invocation sequence. Within the MACCv2 processing-step integration approach, such controlling instance is by itself a "MACCv2-Tool". Figure 4.18 depicts this hierarchical approach. Several nested toolflows are combined at higher level to form a combined toolflow instance. Especially in combination with tool specialization, which provides the flexibility to adapt existing toolflows to upcoming system architectures, such a hierarchical approach is crucial for high reuse of existing optimization and analysis approaches in future developments.

Toolflow construction is closely related to the implementation of individual tools. Even though toolflow tools do not contribute any optimization or analysis technique but solely coordinate invocation of sub-tools, they are regular tool instances. Therefore, the implementation starts with the same steps as for any other MACCv2 tool.

A toolflow requires a dedicated object class to be implemented which is derived from `MACC_Tool` or any intermediate class. Such an implementation includes provision of at least the implementation of the invocation method and some informational items, like name, description, applicable system-model classes and dependencies. Within the invocation method, the major part of the implementation consists of enumerating the required subtools, setting up configuration entries of these tools and invoking them in desired order, eventually, analyzing the intermediate results and taking further invocation decisions.

More sophisticated toolflows may define interfaces and configuration options at the toolflow level. Again, the approach for toolflows does not differ from the interface and configuration definitions described in Section 4.4.3.

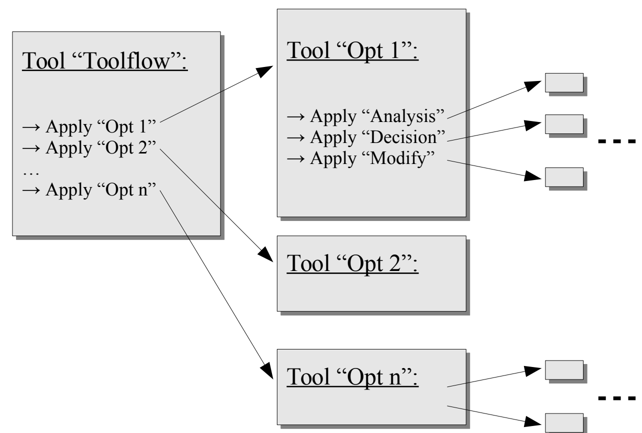


Figure 4.18: Tool hierarchy.

Following this hierarchical toolflow implementation approach, one topmost toolflow instance can be identified for each optimization and analysis approach. To perform the desired actions, the user has to be able to explicitly apply the toolflow to the target-system description. Since the tool-based configuration and invocation methods are well-defined, several options exist on how to access a particular toolflow:

- The MACCv2 framework provides a set of predefined executables which can be used to construct system descriptions according to template models, populate these system models with application code and apply any tool-based processing step, including fullfledged toolflows as described in this section.
- Another framework-provided method targets the case where system descriptions are interactively developed via a graphical editor integrated into the Eclipse-IDE. Within this IDE, a tool invocation interface is provided. This way optimization techniques or other processing steps can be immediately applied to the system description being designed. Examining the results on the graphical representation of the target-system description gives the opportunity to perform interactively several design space exploration iterations.
- Finally, the processing-step invocation can be performed via dedicated user-interface frontends, enabling well-structured presentation of configuration options and easy access to the processing-step results. Especially in the case of a project-driven optimization technique development where a user-centric presentation of achieved results is required, such dedicated frontends are beneficial. Especially, when implementing dedicated graphical frontends, the user-interface abstraction presented in Section 4.2.4 provides powerful implementation foundation for interactive toolflow operation. Using the user-interface abstraction gives the toolflow sufficient control and information on the current state, so non-interactive tool operation modes (i.e. script-based invocations) are not affected.

4.5 Conclusion

The work presented in this thesis tries to overcome typical problems in the development process of source-code optimization techniques. The corresponding models and approaches have been introduced in Chapter 3. Actual implementation details and practical examples have been presented in this chapter.

Basic Services

The set of framework services has been implemented as C++ object classes. A significant set of services is closely related to the common base class. The details presented here show how typically occurring services like reflection, persistent data retention, data annotation and runtime-linking support have been implemented within this framework. Besides this, the common base class offers services which go a step further. Since reuse and self-contained modularity are at the focus of this framework, the implementation of object-reference tracking and auto deletion, cross module event notification, unified access to dedicated class members and object factories has been presented here as well. These implementation details have been accompanied by practical examples which give a good starting point to benefit from these services in optimization technique implementations.

High-Level Services

Based on this foundation, higher-level services have been implemented. The first one, the user-interface abstraction, enables development of interactive optimization techniques. Three aspects of user interaction have been implemented. Optimization techniques may expose dialogs to the user requesting simple decisions or offering a set of options to choose from. Typically, optimizations take prolonged execution times. To keep the user updated on current progress, corresponding user-interface service is offered to the optimization technique. User-interface abstraction also provides means of visualizing optimization technique results. The implementation details of these services have been presented in this chapter. Finally, in the course of implementing the MACCv2 framework services, three user-interface types have been provided. A plain-text-based interface suitable for command-line usage of optimization techniques, a graphical user interface which is best suited for optimization result presentation and interactive usage and a target-system model development-focused interface which has been implemented as an Eclipse-IDE plug-in.

System Modeling

The system-modeling approach as integrated into the MACCv2 framework has been built upon the basic services provided within this framework. Based on the common object class, tight integration with other high-level services has become possible. Especially, the application-code representation is based on the same object-class model which allows for seamless formulation of relations between these representations.

The practical implementation of the system-modeling approach consists primarily of object-class declarations representing particular system-model entities and a set of actual target-platform descriptions. Each target-platform description consists of a platform template and the corresponding set of component and channel-representing object classes.

Since these platform descriptions were implemented in the context of the MNEMEE project, which is targeting memory-aware source-level optimizations, a set of aspect handlers for the most relevant set of system-wide properties has been implemented. Even though the implementation has started off for the MPARM platform, practical experience has shown simple reuse for the other platforms as well.

Successful implementation of real-life target-platform models and usage in projects and teaching, as can be observed in detail in Chapter 5, show feasibility and practical relevance of this system-modeling approach. The decision to base the system-model items on C++ object classes, provides a natural interface within an optimization technique implementation and takes advantage of programming language native class inheritance relations.

Processing-Step Integrating

Actual optimization technique development is supported via the tools implementation service. Tools represent self-contained entities which encapsulate optimization and analysis steps and provide unified invocation and configuration APIs. The implementation details presented in this chapter focus on the steps and effort needed to implement processing steps and how to link those steps into a toolflow.

This framework has shown its practical relevance in the MNEMEE project. The effort for integrating optimization and transformation techniques developed at disjoint organizations across Europe has been significantly reduced. Summarizing the framework-related contribution of this thesis, the design and the set of services provided within this framework can be considered to be a well-balanced foundation for the development of memory-aware optimization techniques.

5 Evaluation Results

The system-modeling approach presented in this thesis is expected to provide a common target-platform description for various source-level optimization techniques. In particular, memory-subsystem-aware optimization techniques are dependent on the presence of a precise system model. Since the approach is part of an optimization technique development framework, there is effectively an unlimited set of optimization techniques which could be developed using this framework, and which may expose different requirements on the target-platform model. In general, a balanced approach between the level of detail, effort to model a target platform and the precision of such models is required.

The system-modeling approach presented in this thesis tries to provide such a balanced implementation. This chapter evaluates the system-modeling approach in the previously-identified dimensions of design complexity, precision and abstraction level. Subsequently a closer look is taken on the benefits arising from application of the MACCv2 framework in a research project and teaching activities.

5.1 Motivation

Quantifying the outcome of a system-modeling approach is a challenging task. In the context of a framework which is expected to support developers in implementing cutting-edge memory-aware optimization techniques, intuitively the question arises how well the proposed system-modeling approach suits this purpose. As indicated, there are several orthogonal dimensions which need to be taken into account to estimate the fitness. Unfortunately, also non-functional aspects contribute to the overall result, preventing a closed form, numeric, fitness or improvement representation as typically used for the results achieved by the optimization technique to be implemented within this framework. Nevertheless, each aspect is analyzed in detail to provide a notion of best-suitable application scenario to upcoming developers. In the case of precision evaluation of system-wide properties computation, a comparison to an actual target platform is possible. Therefore, in this case, a deviation value is calculated.

First, considerations in the process of estimation of properties and applicability in a source-level optimization development flow go towards the set of dimensions to be evaluated.

In the framework-based development process three groups of users have been identified:

- The first group of users has the primary task to provide system-model descriptions. They develop component and channel classes, system-model templates and implement the property computation rules for each component and channel (i.e. energy-consumption or latency models and corresponding aspect handlers). This group of developers is primarily interested in the methods provided by the framework to implement such a system description. In particular, the API definition,

the initial overhead, eventually the set of already implemented components and channels. Summarizing this domain, the system design effort needs to be analyzed.

- Following the typical use scenarios, the second group of developers is considered to provide the building blocks for actual optimization technique development. These are analysis or transformation steps, previously developed optimization techniques or abstractions of external tools (i.e. access to compilers and linkers). With respect to the system-modeling approach, this group uses particular instances of system descriptions. While using system descriptions the focus shifts towards the model and in particular towards the structural level of detail available within a system description.
- Finally, the third use scenario covers the actual development of new optimization techniques. They are typically combined of existing building blocks. Therefore, the appropriate level of detail of the system model is also of increased importance. Nevertheless, typical memory-aware optimization techniques also require precise models describing various system properties. Since the system-modeling approach includes a fast method for requesting such values at runtime in a fraction of time compared to state-of-the-art approaches based on full-system simulation, the precision of provided system properties is crucial for taking founded optimization decisions. Therefore, system-model precision is analyzed as the final dimension.

A system-modeling approach on its own will provide only theoretical benefit to the research community. Therefore, it has been implemented and embedded into a fullfledged memory-aware optimization development framework. The second part of this chapter focuses on the framework benefit evaluation. The work presented in this thesis has found real-life application in an European Commission funded project and several embedded systems teaching activities. This provides a good foundation to evaluate the framework benefits along these corresponding application scenarios.

Since typical optimization techniques are preferably not going to be developed from scratch, but based on previous work, the final section evaluates aspects of porting already existing optimization techniques into a MACCv2-based implementation. An exemplary memory-aware optimization technique is presented and corresponding transformation steps are analyzed according to reuse and implementation effort.

5.2 System Design Effort

The effort for system-model design is primarily driven by the effort for implementing system-model building blocks and the system-model template. In both cases APIs provided by the MACCv2 framework are used. Since the effort grows with the complexity of modeled target platform, an estimation of the growth in relation to system-model size in terms of number of components and channels is performed.

As a starting point, a basic system description consisting of a single processor, a memory component and an interconnecting bus is evaluated for its minimal design effort. Figure 5.1 shows an example of such a system model.

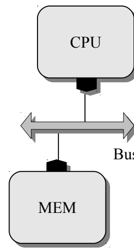


Figure 5.1: Initial system model.

Component Modeling Effort

Provision of system-model descriptions occurs with some initial effort. Mainly the implementation of an object class derived from the component or channel base class, or any further specialized one (i.e. a processor or memory class) contributes to this effort.

Implementing such derived classes exposes only a moderate initial effort. Since typically, the primary purpose of a component or channel class is to provide a unique object type for each system-model item, this effort requires only regular C++ programming skills, namely, declaring an object class and implementing the constructor member function. From the framework perspective, the task of defining system-model components and channels is supported by various macro definitions, effectively reducing the effort to few lines of code. Typically, this includes the definitions required for persistence, providing component descriptions and defining relevant address spaces. The unavoidable one-time effort with respect to the framework service is to gain an overview of the methods and services provided by the MACCv2 framework. Nevertheless, even for these developers who prefer a jump-start approach or evaluate the fitness of this framework for their projects, documented example component definitions are provided. Therefore, it is a valid conclusion, to quantify the effort for system-model component and channel definitions to be very moderate. A real-life example supports this conclusion. The minimalistic host system model defines a system similar to the example in Figure 5.1, consisting of two component classes (processor and memory) and one interconnect (a bus). None of these definitions exceeds 100 lines of code in total, including excessive comments, and formating.

Aspect Modeling Effort

In addition to these minimalistic definitions which are sufficient for plain structural system models, a typical system model is going to use the unique system-wide property computation feature of this framework, as presented in Section 3.4. Such system-wide properties require the presence of corresponding handlers at each component and each channel. Providing such handlers is a two step approach: Defining a particular handler type and binding such a handler to the component or channel object. The second step is basically limited to the instantiation of a handler object of the appropriate type and

adding the reference to the set of handlers. Typically, such manipulation requires only up to two additional lines of code per handler added to a component or channel.

The actual effort for defining handler object classes depends on the complexity of the underlying model to be implemented. Besides the typical class definitions, only the computation method needs to be implemented. For commonly-occurring cases (i.e. accumulating constant values) the framework provides predefined handlers. In such cases there is no need to define target-platform-specific handlers. Even when no suitable predefined handler exists, a typical user-defined handler class requires about 60 lines of formated and documented code. The processor energy-consumption computation handler for the MPARM system model is a real-life example for such an implementation.

Top-Level Design Effort

According to the system-modeling approach, a system model consists of, besides the component and channel items, a top-level system-description object class. Similar to the component definition, the initial purpose of the system object class is provision of a unique class for that particular system-description type. Therefore, the initial effort is similar to the one expected for component and channel definitions. According to the lines of code metric, the definition of a system class is typically less than 100 lines. A practical example at this complexity level is the host system model. The top-level system object class definition requires 80 lines of code, including formating and comments.

MPSoC Design Effort Scalability

Continuing the design effort considerations for a typical system model, the next step goes towards the design of multicore system models. The effort for a basic single core system model as considered previously in this section, is taken as the starting point and the baseline. Depending on the general system type, the effort for the component and channel definition may vary. A multicore system model consists in general of function blocks which are connected to shared resources via system-wide interconnections (i.e. a bus). The design effort for a function block is considered to be equal to the effort for the single core system presented in the beginning of this section. In addition to this, the system-wide interconnection channel and typically corresponding bridging components need to be implemented. This is a one-time effort per system-model type. Per-component or per-channel implementations remain typically within the 100 lines of code range, found in the single core models. In homogeneous target platforms each function block is of the same structure. Therefore, regardless of the number of function blocks present in the system model, no additional components or channels need to be defined. Consequently, no additional implementation effort arises. In heterogeneous target platforms each function block may have a different set of components and channels (i.e. differing types of processing cores). In this case, each function block may require a similar effort as the single core system model. Therefore, the overall system-model design effort grows in the worst case linearly with the number of function blocks present in the target platform to be modeled.

Template Definition Effort

Finally, a typical system model provides a target-platform template. The purpose of such a template implementation is the definition of a typical target-platform system model. Primarily, it is used in the automatic construction of system-model descriptions for unattended optimization run, or as a starting point for the GUI-based design of target-platform system models.

Designing such target-platform templates consists basically of instantiating the component and channel objects, linking their ports according to the target-platform structure and implementing corresponding address space mapping rules. Each of these steps typically consists of a single digit count of lines of code. Defining the mapping rules can be considered the highest effort step within the target-platform template definition. Depending on the memory map established in the target platform, in the worst case each segment in the memory map requires a mapping rule entry on each traversed port. Fortunately, in typical MPSoC platforms several segments can be combined into one mapping rule. In a typical scenario each function block requires at most two mapping rules. The first one covers the range of tightly coupled memories, the second combines all ranges which require access to the secondary-level bus. Further selection of particular target components occurs in the mapping rules for these components (i.e. shared memories, I/O devices). Since these rules are common to all function blocks, defining these mappings is a constant effort regardless of the number of function blocks present in the target platform.

In total, the effort for a target-platform template definition can be considered to be moderate. The minimalistic host system-model template requires less than 200 lines of code, while a fullfledged, configurable MPSoC system-model template for the CoMET ARM platform currently requires about 500 lines of code.

Concluding the overall system-model design effort, several effort contributing tasks have been identified. Component and channel design, implementation of aspect handlers defining the component's contribution to system properties and, finally, the definition of system-model templates connecting these components and channels in a preferable order. Combining these individual task efforts into an overall system-modeling effort, shows a linearly growing complexity in terms of distinct component and channel types. Each component or channel implementation contributes a fixed amount of lines of codes (independent of other components and channels). The system-model template requires for each component port a single digit count of lines of code. Since the number of ports a component exposes is also fixed and independent of the presence of other components, the overall increase in lines of code in system template definition due to additional components or channel types is linearly related to the number of such components or channels. Therefore, the overall system-modeling effort shows a linear and moderate growth in relation to the number of component and channels.

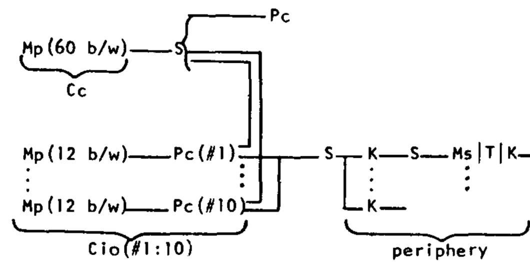


Figure 5.2: Exemplary Bell/Newell PMS representation of a CDC 6600 system [7].

5.3 Abstraction-Level Considerations

A reasonably applicable system-modeling approach faces several contrary requirements. Developing system models should be simple, while maintaining sufficient flexibility to be able to model a wide range of target-platform types, including future platforms. The previous section shows that in terms of design complexity, a balanced approach between complexity and flexibility has been achieved. Similar contrary requirements exist for the choice of preferred abstraction level. Higher abstraction levels with less detail reduce the modeling effort, but may also diminish the optimization potential. Defining system descriptions at a too detailed level, may obscure valuable structural properties, and therefore effectively also reduce the possible optimization gain. The decision for a particular abstraction level influences the extend of the domain of target platforms feasible to be modeled in such a system description. Therefore, a balanced approach has to cover as much as possible of the relevant target-platform types while providing appropriate level of detail.

According to Chapter 3 and the overview of various system-modeling approaches in Chapter 2, this system-modeling approach is located at the processor-memory-switch level as introduced by Bell and Newell [6]. Decomposing system-level designs into these building blocks gives valuable structural information for memory-aware optimization techniques. On the other side PMS models are sufficiently abstract, not to limit the set of possible components. Compared to the adjacent abstraction levels, the PMS level provides the best balance between modeling effort, level of detail and covered range of possible system models. Especially, towards lower abstraction levels, algorithmic-level models, as described by Hanna and Melham [67], require detailed behavioral and instruction set descriptions, which are not needed for memory-aware optimization techniques and thus just increase modeling complexity.

Figure 5.2 shows a typical representation of a system according to Bell and Newell. Observing the structure shown in this figure, a straightforward relation to a MACCv2 model can be concluded. There are components connected to each other via links. Bell distinguishes several types of components which match well to the MACCv2 component and channel representation. In particular:

- Memory (M), a storage component, has the same dedicated component type in MACCv2
- Link (L), a connecting component. Within MACCv2 the corresponding item is a channel.
- Control (K), an active component. Within MACCv2 this type would translate into any component type which exposes at least one initiator port. The most common example would be a DMA unit.
- Switch (S), a switched link component. This relates also to the class of MACCv2 channels. More precisely, to the Bus channel class.
- Transducer (T), an I/O component. MACCv2 has no predefined I/O component class. Basically these items can be matched to the basic component class.
- Data-operation (D), a data manipulation component. Similar to transducers, MACCv2 does not provide a dedicated component subclass for this type of components. Nevertheless, it is feasible to map them to the basic component class.
- Processor (P), the processing unit. MACCv2 offers a processor component, which is capable of maintaining the relation to corresponding application code.

The inverse mapping can be deduced from the previous item list. In general MACCv2 distinguishes between direct links and bus-based channels. The corresponding representation in PMS models would be links (L) and switches (S). Memory components and processing units have a direct representation in PMS notation. Components which can initiate memory accesses, are best mapped to Control (K) components, since they are the sole active part besides processing units. The remaining set of target components will map either to a memory-controller part, or a data-operation controller part, depending what is its intended functionality.

With this mapping, the MACCv2 system description defines a target-platform representation which can be considered to be located at the PMS abstraction level. This representation fits well the level of detail required for memory-aware optimization techniques. Similar to the representation proposed by Bell and Newell, components and channels may have annotated properties which increase the level of detail as necessary. In addition to the plain PMS target-platform representation, MACCv2 models expose a formalized notion of access routing. A link between components includes a description of the relation between adjacent addressing schemes. This allows for automatic traversal of MACCv2 system models, and computation of system-wide property values.

5.4 Target-Platform Representation Precision

Knowledge of target-platform properties is important for source-code optimization techniques to be able to adapt the code to a more efficient usage of target-platform hardware.

According to the optimization targets, the expected result is reduced energy consumption and/or reduced runtime.

There are several approaches to incorporate such knowledge into the optimization technique. The least preferable, nevertheless often occurring, is to develop the optimization technique based on implicit target-platform assumptions, being hard coded into the optimization technique implementation. Typically, such approaches enable immediate access to target-platform properties at optimization technique runtime.

Moving towards retargetable optimization techniques, a common approach is usage of target-platform simulators or the actual hardware as a black-box for evaluation of the benefit achieved in a presumably optimizing code transformation. Since such evaluations are time-consuming, and are separated from the actual optimization decisions, they influence such decisions only indirectly in an iterative approach. Nevertheless, due to the whole-system evaluation the estimated results are precise.

The approach presented in this thesis tries to combine advantages of both approaches, enabling development of truly platform-independent optimizations which, nevertheless, may take platform properties at runtime into account. For a detailed description refer to Chapter 3. The key benefit of this approach is the possibility to provide system-wide property values based on a component or channel wise locally-defined data set which is combined on request into such global values. Typically, such properties are not limited to, but preferably used to, describe energy-consumption-related or runtime-related properties.

Within this section the precision of these system-wide properties is evaluated and compared to full-system-simulation results. The MPARM platform and the corresponding simulator are used as a baseline. The advantage of this platform is the possibility to evaluate both energy-consumption and runtime properties simultaneously on the same platform. The simulation platform exposes a well-defined, community-accepted [68] energy and runtime model for an ARM-based MPSoC platform. To achieve improved configurability of the memory subsystem the simulator has been combined with the memory hierarchy simulator MEMSIM [42]. The system configuration defines one processing tile with a 12k bytes scratchpad and a 12M bytes main memory connected via an AMBA-AHB bus. The energy and latency values for these memories are configured according to the reference design:

- Scratchpad memory read: 0.0241275682 nJ / 0 wait states (WS)
- Scratchpad memory write: 0.0080780647 nJ / 0 WS
- Main memory read: 10.6813104793 nJ / 10 WS
- Main memory write: 1.0667890999 nJ / 10 WS

Figure 5.3 shows such a typical MPARM platform using the default configuration of four cores. According to the minimal setup, caches have been disabled. Since the platform exposes a homogeneous structure, the evaluation is performed for the first processing core without loss of generality.

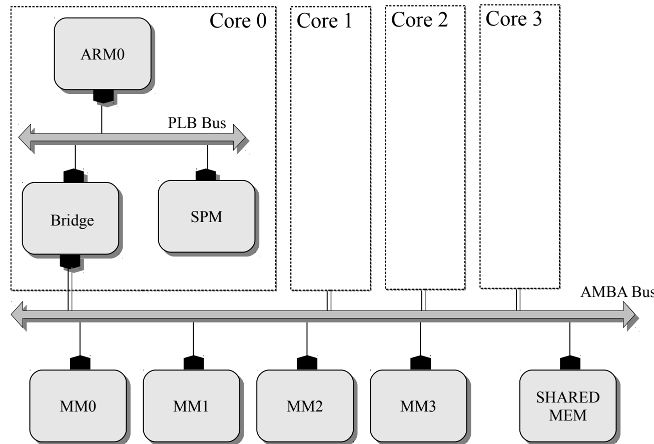


Figure 5.3: MPARM target platform.

The MACCv2 system-model description follows the structural properties of this target platform. Except for the omitted I/O devices the set of components and channels matches exactly the items of the target platform. To focus the evaluation approach on memory-aware optimization techniques targeted in this thesis, an exemplary scratchpad allocation technique is used next.

The static scratchpad allocation used for demonstration purposes in this thesis was presented by Steinke et al. [69]. Further extension toward multiprocessor systems has been done by Verma et al. [43]. Basically the approach solves a knapsack problem for each processor separately using integer linear programming. Without loss of generality, this example targets energy reduction as the objective. A set of input parameters is required per processor to formulate the ILP equations:

- A set of code or data items which can be placed independently in the main memory or the scratchpad memory. These items are identified as memory objects in literature. In our example global variables (i.e. scalars and arrays) and functions are used.
- The size of each memory object.
- The access counts to each memory object.
- The scratchpad size as the limit for the knapsack.
- The energy consumption per main-memory access.
- The energy consumption per scratchpad-memory access.

Subsequently, the approaches to gather required data in the MACCv2-based environment are presented. Starting with the set of movable items, they can be identified easily in the abstract syntax tree of the C program representation within the MACCv2 system description. The processor nodes in the system model incorporate the ICD-C-based

representation of application program code intended for execution at this processor. Iterating over the global symbol table provides names and references to these memory objects. The sizes of these memory objects are also accessible via symbol properties stored in the application-code representation. According to the list of optimization technique prerequisites, next, access counts for each memory object are required. To gather such values, the scratchpad optimization technique will invoke a corresponding MACCv2 tool or specify a dependency on such tool asking the framework to run this analysis step in advance. The result will be annotated to the system description, in particular to each symbol within the application-code representation. Therefore, from the point of view of this example optimization, the access count information is inherent to the system description, and its construction may vary depending on the target-platform model. Currently, access counts to each of these memory objects are generated via profiling. Once static analysis methods become more sophisticated, they can seamlessly replace the profiling step.

In the context of this evaluation, steps required to find the architectural properties are of higher relevance. Determining them is typically as easy as looking up some table-based predefined values, but with the confidence of always getting precise up-to-date values, without the need for manual adaption to new platforms. Refer to Section 3.4 for more details on modeling and retrieval of these system-wide properties.

To be able to construct the ILP formulation, the size of the scratchpad memory must be known, or more precisely, the size of the fraction of this memory mapped into the processor address space. In typical platform designs, including this MPARM platform, there is no subdivision of memories, and each processor has either access to the entire memory or no access at all. Nevertheless, to achieve a generally valid approach, retrieving the scratchpad-memory size is a two-step approach. In the first step, the MACCv2 framework-provided address space analyzer is applied. The result is a mapping list of address ranges in the address spaces of an initiator component (i.e. an ARM processor) and the final target component. The analysis is performed by default across all memory-hierarchy levels, therefore in the mapping the final target component (i.e. main memory or scratchpad memories) will be visible. For this optimization technique the memory layout, as observed by a processing unit, is the interesting outcome of this analysis step. In this example, the first processor in the MPARM target platform shows this mapping:

```
ARM0.[0x0-0xbfffff] -> MM0.[0x0-0xbfffff]
ARM0.[0x19000000-0x190fffff] -> SHM.[0x0-0xfffff]
ARM0.[0x22000000-0x22002fff] -> SPM0.[0x0-0x2fff]
```

In addition to the mapping ranges, the type of targeted memory - in general, the type of a targeted component - is of importance for an optimization technique. Since the MACCv2 framework-based system descriptions define dedicated object classes and corresponding inheritance hierarchy for each component type, optimization techniques requiring this type knowledge, can easily iterate over the set of components and examine each item for the relevant class membership. In the case of the optimization technique used in this section, the presence of scratchpad memories is of importance. In

the MACCv2 system model, scratchpad memories can be identified as components being derived from the specific scratchpad-memory class.

Once all relevant memories are identified, a second step has to be performed to determine the size of allocable memories. Simple arithmetic operations on the address ranges are sufficient to compute a scratchpad size of 12k bytes according to this example architecture.

The energy values, which are required to compute the achievable gain for each allocable memory object, can be retrieved in a similar way. As described in Chapter 3, the key feature of this model is to provide system properties via a query-based interface. Two sets of queries are required for this optimization. The first one would be placed to the address space range of the main memory, the second one to the range of the scratchpad memory. Each set would collect three values; for data read, data write and instruction fetch. The difference between corresponding values of each set is the gain factor multiplied by the access counts.

These values complete the set of prerequisites for the ILP formulation. Solving the ILP will result in a set of decision variables indicating which memory object has to be placed on the scratchpad. The key value directing the ILP solution is the gain which can be achieved per memory object when moving it to the scratchpad memory. Therefore, the evaluation of MACCv2-provided values is performed in relation to the values determined in the whole-system simulation.

The final missing part is the actual application code to be optimized. To be able to manually verify the results, a reasonably simple application code has been chosen. The selected application code performs a chain of matrix operations. In the first step, a matrix $\boldsymbol{A}$ has to be transposed, and in the second step, a second matrix $\boldsymbol{B}$ is added and the result is stored in another matrix $\boldsymbol{C}$.

$$\begin{aligned} \boldsymbol{AT} &= \boldsymbol{A}^T \\ \boldsymbol{C} &= \boldsymbol{AT} + \boldsymbol{B} \end{aligned}$$

All matrix values are 32 bit unsigned integers. Two sets of experiments were performed, one with 10×10 matrices, another with 30×30 matrices.

In the context of this evaluation, the selected benchmark has the advantage of having a fixed number of accesses to each matrix which can be determined via profiling. Table 5.1 shows the access counts for 10×10 matrices which were determined in advance.

Matrix	Reads	Writes
$\boldsymbol{A}$	100	0
$\boldsymbol{B}$	100	0
$\boldsymbol{C}$	0	100
$\boldsymbol{AT}$	100	100

Table 5.1: Access counts for 10×10 matrices.

Corresponding values for 30×30 matrices are shown in Table 5.2.

Matrix	Reads	Writes
A	900	0
B	900	0
C	0	900
AT	900	900

Table 5.2: Access counts for 30 × 30 matrices.

To formulate the baseline, the energy consumption caused due to the execution of matrix operations on the MPARM system has to be determined. For each matrix size, the benchmark was run with all matrices located in the main memory of the MPARM simulator. The total energy consumption in this case was 27130.266 nJ for the 10 × 10 matrix and 231908.203 nJ for the 30 × 30 matrix. Afterwards, each matrix was moved one-by-one to the scratchpad memory. Since the scratchpad memories are more efficient in terms of energy consumption, a lower total energy consumption was determined. The resulting energy-consumption decrease is summarized in Table 5.3. These values are the gain factors which could be used in the ILP formulation when solving the knapsack problem related to an optimal allocation of memory objects.

Matrix	10 × 10	30 × 30
A	1122.741 nJ	10104.672 nJ
B	1122.741 nJ	10104.672 nJ
C	164.459 nJ	1480.125 nJ
AT	1287.202 nJ	11584.812 nJ

Table 5.3: Measured gain per matrix.

Collecting these energy-consumption gains in the way described up to now represents state-of-the-art approaches often found in literature. Observing the effort and runtime of this task it may contribute a significant share of the overall optimization technique runtime. The approach presented in this thesis tries to avoid this effort and provides such values in a fraction of the time without use of full-system simulators. Computing such values has to be sufficiently precise to gain acceptance in future optimization techniques.

Up to now, the baseline values for the comparison have been obtained. Subsequent steps describe how to compute the same per-matrix energy gains using MACCv2-provided methods. An important prerequisite is a well-defined system model. The system model used for this comparison has been designed according to the MPARM simulator setup. The energy and latency values for memories are configured to the same default values as used in the simulator. The interconnection energy computation is based on work done by Bona et al. [70]. The subsequent values denoting calculated total energy consumption per access were retrieved by four access queries. Two queries are directed to the scratchpad memory and two to the main memory, each one for a 4 bytes read and a 4 bytes write access, corresponding to the size of the matrix elements.

A typical query consists of few lines of code only: setup, add aspects, perform query and retrieve value:

```
MACC_Access *acc=new MACC_SingleAccess(...);

acc->addData(ASPECT_ENERGY);
acc->addData(ASPECT_CYCLES);
acc->queryAccess(AS_WORSTCASE,ASPECT_ENERGY);

val=acc->getValue(ASPECT_ENERGY);
delete acc;
```

Matrix	10×10	30×30
A	1120.457 nJ	10084.114 nJ
B	1120.457 nJ	10084.114 nJ
C	164.238 nJ	1478.143 nJ
AT	1284.695 nJ	11562.258 nJ

Table 5.4: MACCv2 computed gain per matrix.

Via these queries, total per-access energy values of 0.0791284 nJ per scratchpad read, 0.0630784 nJ per scratchpad write, 11.2837 nJ per main-memory read and 1.70546 nJ per main-memory write have been retrieved. These values multiplied by access counts result in the gain value for each matrix. These values are summarized in Table 5.4. Comparing these values to the ones generated via full-system simulation results in the conclusion that the system-modeling approach presented in this thesis has the capability to provide very accurate information regarding the energy-consumption estimation. The computed results were all less than 0.21% off compared to the actual values. Furthermore, the results are stable along the increased number of accesses, even for the bigger matrices, having nine times higher access counts, the deviation remains almost the same.

Matrix	Deviation
A	0.203%
B	0.203%
C	0.134%
AT	0.195%

Table 5.5: Relative deviation from simulation results

5.5 Application Scenarios

The MACCv2 framework and the platform description approach presented in this thesis have found application in an European Commission funded project and several embedded systems teaching activities. Following subsections present the benefits resulting from the

application of this work in both areas. Furthermore, the path towards exploitation of MACCv2 framework even for already existing memory-aware optimization techniques is sketched in the last subsection.

5.5.1 Application in MNEMEE Project

The tight integration of several optimization techniques into one cooperative toolflow has been highlighted in the project review process. Especially, this cooperating operation of each optimization step leads to the noticeable energy and runtime saving presented in the project introduction in Section 2.7.3. The MACCv2 framework provides two key features which served as the foundation for this tight integration. The first one is the system-modeling approach. The second is the processing-step interaction and encapsulation service. In following sections the benefit and contribution to the MNEMEE project are evaluated.

5.5.1.1 Target-Platform Architectures

The MNEMEE toolflow has been designed to be applicable to various target platforms. In the course of this project, several platforms haven been chosen to apply the toolflow in the first place. This implies the presence of corresponding platform descriptions which provide metrics and structural properties for each platform. To achieve this goal the MNEMEE toolflow uses the system-modeling facilities provided by the MACCv2 framework. The following set of target platforms has been used in the MNEMEE project. For each platform a short introduction, followed by a description of corresponding MACCv2 system-model implementation, is given.

Host platform architecture: The MNEMEE project uses the host-based application-code execution and corresponding system model. The system model aims at providing an abstract compilation and execution platform. A typical setup consists of a single core and a single main memory. Figure 5.4 depicts such an abstract target-platform architecture. Typically, such platforms are used for profiling steps. Target-platform-independent values (i.e. variable access counts) can be retrieved in this environment within significantly shorter time compared to a full-system simulation. The MACCv2 host platform system description embeds the native runtime environment of the host platform, effectively enabling the same handling of application code and invocation of optimization and transformation tool as for the actual target platform.

MPARM target-platform architecture: Initially the MPARM platform has been developed and used as the driving demonstrator for the system-modeling approach presented in this thesis. Besides this, the same platform model has been used in the MNEMEE project as the primary platform for evaluation of the toolflow developed in this project. Several refinements have been contributed within the MNEMEE project. Basically, an improved configurability of memory layout and energy metrics were implemented.

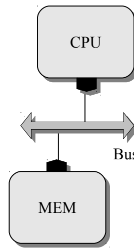


Figure 5.4: Abstract host architecture

The general structure has already been depicted in Figure 5.3 in Section 5.4. Typically, a standard four core system is modeled. The memory subsystem consists of local scratchpad memories, private main memories and a common shared memory. Typical sizes are 12k bytes scratchpad 12M bytes main memory and 1M byte share memory.

To deviate from these defaults, all memory sizes can be configured on system-description instantiation. Furthermore, the number of processing cores can also be adjusted.

In the context of the MNEMEE project, the MPARM platform has the advantage of providing a PC-based simulator which includes energy-consumption estimation and a cycle-accurate runtime estimation. Both metrics have been vital for the work performed within the MNEMEE project. Since the optimization techniques are expected to perform code transformations resulting in reduced energy consumption and reduced runtime, an evaluation platform providing these metrics is mandatory.

According to Section 5.4, the system model of the MPARM platform implemented in the MACCv2 framework achieves a highly precise representation of both relevant system-wide properties, namely per-access energy consumption and latencies. Summarizing the key properties, the MPARM platform offers a community-accepted energy and runtime estimation on a cycle-accurate full-system simulator combined with a precise system-model representation usable within MNEMEE optimization techniques. It is a valuable foundation for further development and academic evaluation of these optimization techniques.

CoMET target-platform architecture: Unfortunately the MPARM platform has a significant drawback. The full-system simulator operates only at a moderate speed. Therefore, real-life application code as optimized within the MNEMEE toolflow, has significant evaluation times. This lead to the decision to introduce another target platform, which could be used in the development process to achieve faster results. This target platform was primarily designed to resemble the MPARM platform on a faster simulator. The CoMET simulator [62] is an industry-approved simulation environment. The simulation environment has been configured to implemented a target platform which is structurally comparable to MPARM. Furthermore, additional variations of the structure can be configured at instantiation. Basically, there are two CoMET-based target-platform simulation setups. The first resembles exactly the MPARM memory-subsystem structure. According to Figure 5.5 this is the so-called flat target platform, where each private

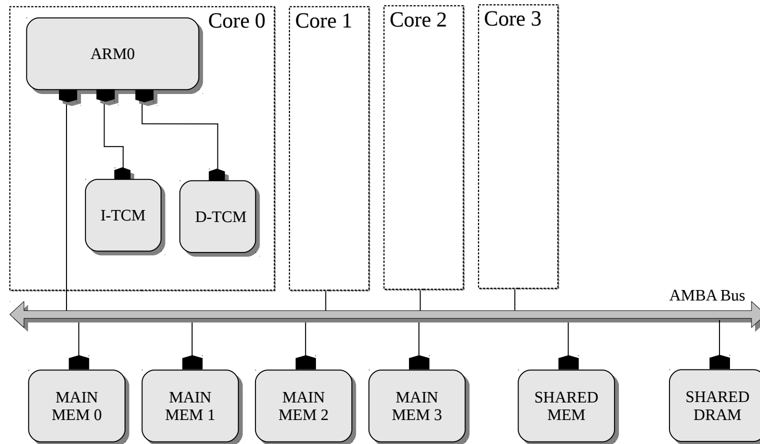


Figure 5.5: CoMET flat target-platform architecture

main memory is connected to the system-wide bus. Typically, this results in increased bus congestion when multiple processing cores access data or execute code from their local main memories.

Another CoMET-based simulation platform has been designed. This platform was intended to evaluate the effects related to moving the main memory closer to the processor core. Figure 5.6 depicts such a platform. In this case, local main memories are connected to their corresponding processing cores via dedicated buses. In this setup no bus congestion due to main-memory access can occur. Only accesses to the shared memory have to pass via the system-wide bus.

Since the CoMET and the MPARM platform expose similar structural properties the MACCv2-based system model for this target platform has been derived from the MPARM model. The scratchpad memories are bound to their corresponding processing cores in a different way. Basically, each uses a dedicated interface to connect to the core. Regarding the different simulation platform setups, MACCv2 system description resembles the flat target-platform model as depicted in Figure 5.5. Since the CoMET platform has no established energy and runtime values, the energy and runtime model of the MPARM platform has been adapted and used for the CoMET target platform.

MSC8144 target-platform architecture: The MNEMEE project has been planned in tight cooperation with industrial partners. One goal was to apply the optimization and transformation techniques developed in this project to real-life application code and corresponding target platform. Such a platform is the Freescale MSC8144. Figure 5.7 depicts the overall structure. It is a four core high performance DSP architecture. The memory subsystem has a three-level hierarchy. A peculiarity of this platform is the capability to distribute dynamically the memory at each hierarchy level among the processing cores. Due to the cross-bar-based interconnection, the memory shares are accessible simultaneously from each DSP. For this platform a physical execution environment was available.

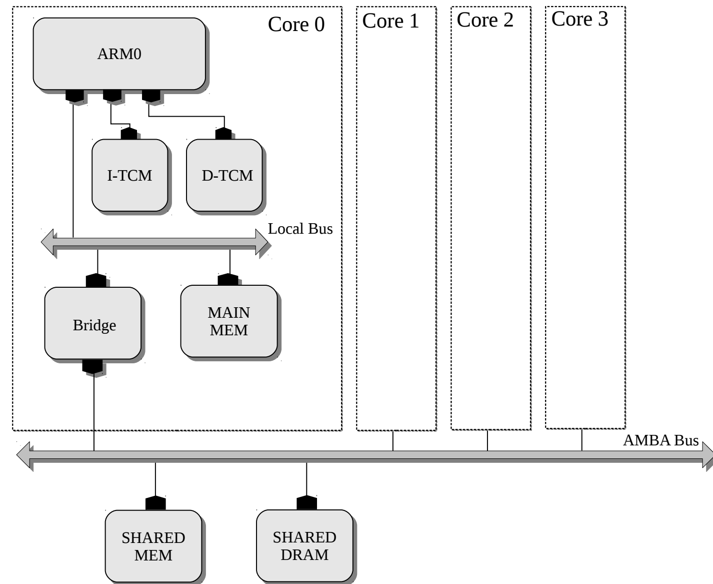


Figure 5.6: CoMET hierarchical target-platform architecture

The system model for the MSC8144 platform has been designed primarily for the application of MNEMEE-provided optimization techniques. Primary focus has been put on well-matching and highly configurable memory-subsystem representation. Therefore, the numerous peripheral devices were of lower importance and have been omitted. Further, the opaque CLASS interconnect has been modeled as distinct interconnections for each level. Based on this structure, the distribution among private and shared memory regions for each hierarchy level can be configured individually. In addition to the memory distribution, parameters of the energy and latency model can also be adjusted. Hardware platform-based measurement of energy consumption (i.e. by the means of an approach as presented by Steinke [69]) could be used to derive matching system-model values. Figure 4.14 in Section 4.3.1.1 depicts more details on the implemented MACCv2 system description.

5.5.1.2 Shared System Model Benefits

The system-modeling approach presented in this thesis allows for a uniform target-platform representation across multiple optimization or processing steps. This ensures a common notion of target-platform properties. Especially cooperating optimization techniques can control their effort to avoid contradicting decisions, which would diminish the overall optimization result. Such a coordinated execution of optimization techniques is expected for the MNEMEE toolflow. According to the toolflow description, a set of memory-aware optimization techniques addressing a more efficient exploitation of MPSoC platforms has to perform its task in a coordinated way. The shared system model provides target-platform parameters for each optimization technique. Typically, each

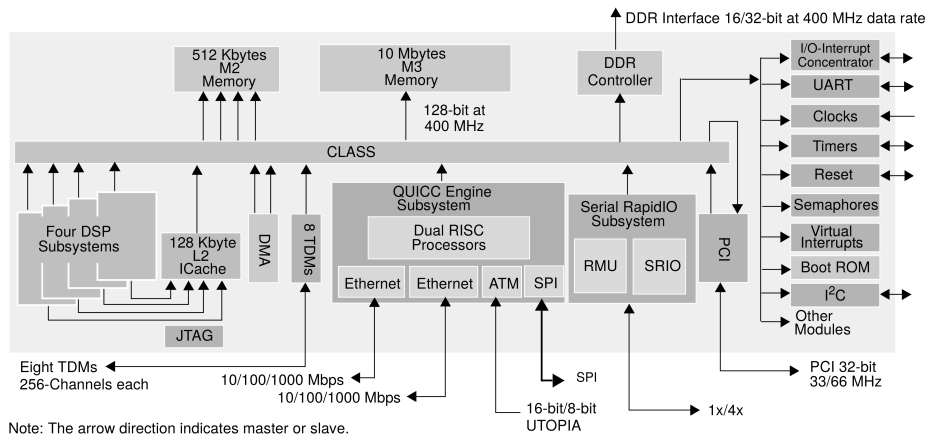


Figure 5.7: MSC8144 Quad core DSP architecture [63]

optimization technique has a different focus on the set of parameters required for its operation. The granularity of requested target-platform properties increases along the optimization chain. At the initial steps (i.e. dynamic data type refinement or parallelization) only structural details are at focus. This includes memory sizes and number of processing cores. At the opposite end of the optimization chain, a fine-grained notion of data access effort in terms of energy-consumption or latency values is required to take scratchpad-memory allocation decisions.

The subsequent matrix in Table 5.6 summarizes the set of requested information per optimization step.

Due to the diversity of information requested by the set of optimization techniques, the system model needs to be sufficiently generic to incorporate all kinds of such properties. Especially, the often observed ad-hoc approaches which combine several data tables describing individual target-platform properties, suffer from inconsistency and a non-uniform access pattern to each property.

The MACCv2 system modeling has the ability to serve a rich set of system properties. The MNEMEE project benefits from this uniform system-modeling approach. First of all, the system-modeling approach is accompanied by a versatile API for accessing these system-model properties. The MNEMEE project tools use this API to retrieve relevant data. Since relying on a present API saves implementation and testing effort, the first benefit can be concluded as a reduced implementation overhead.

Subsequent benefits result from unified target-platform representation. Any optimization step retrieves exactly the same target-platform data. This is a valuable prerequisite which helps in avoiding contradicting optimization decisions. In the case of MNEMEE an additional benefit arises due to the toolflow structure. The optimization steps in MNEMEE are implemented as a two-step approach. The first step implements the optimization logic and records particular optimization decisions. The second step implements code transformations which carry out these decisions in a runtime-environment-specific

	No. of Cores	No. of Memories	Memory sizes	Data Access Properties	Frequency Tags	Memory Tags
DDTR		✘	✘			
DMMR			✘	✘		
TGE	✘	✘			✘	
MPMH			✘	✘		✘
MMAP	✘	✘	✘	✘	✘	✘
SMAP	✘	✘	✘	✘	✘	✘
SPM			✘	✘		✘

Table 5.6: System-model information usage in MNEMEE

way. This separation of processing steps simplifies porting these optimization techniques to different runtime environments. Nevertheless, this introduces additional challenges in keeping these steps synchronized. Implementing such a twofold approach based on the MACCv2 framework relieves the designer from implementing communication and synchronization methods between these steps. Coupling optimization and transformation steps based on MACCv2-provided annotations can be realized in a quite precise but still simple way. The annotation approach allows for attaching complex data structures to any item of the system model. Furthermore, such data may incorporate references to other system-model items. In general, these are typical building blocks for any kind of optimization result representation. MNEMEE exploits this annotation-based communication approach between processing steps at various levels. Passing parallelization decisions to the MPMH step for implementing corresponding synchronization directives or expressing mapping decisions are only two examples for such a system-model annotation-based processing-step communication in MNEMEE.

Finally, the long term benefit of founding the MNEMEE work on the MACCv2 framework results from the well-encapsulated target-platform description. Introducing this optimization flow to upcoming platforms becomes possible at a minimal expense of definition of such a target-platform system model. A practical example for such an optimization technique retargeting has already been observed in the course of this project. While development of optimization techniques and integration into a uniform toolflow have been performed initially for the MPARM platform, the final evaluation of industrial grade application code has also been performed on the MSC8144 target platform. After the toolflow and the optimization technique development process has reached a

mature state, the applicability to the MSC8144 target platform could be achieved by simply providing the appropriate target-platform system model. Similar modeling effort is expected once support for further architectures becomes necessary. Concluding this final benefit, the MACCv2 framework-based system modeling delivers retargetability to the MNEMEE toolflow at minimal implementation effort.

According to these benefits, the MACCv2 framework and the unified system model in particular improved the development process and provided the expected added value in the MNEMEE toolflow. Especially, a significantly reduced implementation effort combined with improved retargetability can be observed. Due to the complexity of memory-aware optimization techniques, the approach of decomposing such optimization techniques into a set of sub-steps is often observed in literature. Furthermore, the domain of memory-aware optimization techniques still offers a wide range of challenging tasks. Therefore, the MACCv2 framework is not bound to the MNEMEE toolflow, but will likely provide similar benefits in future optimization toolflows in the domain of memory-aware optimization techniques.

5.5.1.3 Common Framework Benefits

Besides the system-modeling approach, MACCv2 provides a fullfledged optimization technique integration framework. For the MNEMEE project this framework has been used to bring the project's individual optimization steps into a coherent toolflow.

The MACCv2 framework provides solutions for various aspects of such an integration endeavor, starting from methods for optimization technique encapsulation, followed by methods for inter-processing-step communication up to user-interaction abstraction layer. Hierarchical grouping of processing steps enables high reuse and effort-balanced retargetability of implementations. The seamless integration with the target-platform system model is a unique feature of this framework. Based on this foundation, the actual implementation of memory-aware optimization techniques could be performed at a significantly reduced effort. Further, the common interchange format enables easy plug-and-play of various processing steps. This includes store-and-replay scenarios, where intermediate results are stored to disk and subsequent processing steps are applied multiple times with different parameters. In particular, also variations of the toolflow can be explored, as occurred in case of the mapping step.

These MACCv2 framework features have been used in MNEMEE project:

- Optimization and processing-step representation in terms of MACCv2 tools: According to Section 4.4, the MACCv2 framework offers methods for processing-step encapsulation and interfacing. In MNEMEE, the top-level toolflow as well as sub-processing steps within a particular optimization technique are represented in terms of MACCv2 tools. Especially, the following tool representation features were beneficial for the MNEMEE project:
  - Tool encapsulation: Processing steps are represented as self-contained entities exposing a well-defined invocation interface. One practical example is

the seamlessly interchangeable mapping approach. Since both MNEMEE-provided task mapping methods are implemented as MACCv2 tools and both expect the same system-model annotations, the selection of either memory-aware or scenario-based mapping is only a mater of requesting a particular tool instance in the top-level toolflow.

- Tool abstraction: The MACCv2 framework provides a method for expressing processing-step specialization relations. Therefore, requesting some particular transformation or optimization technique can be performed based on a generic class, while at runtime the framework ensures the usage of the most appropriate instance for a given system-model type and runtime conditions. This feature has been used excessively in the MNEMEE toolflow for interfacing with target-platform compiler and linker tools. Depending on the target-platform model, the MACCv2 framework provides the right compilation and linking tools for that particular platform.
- Tool hierarchy: Decomposing optimization or code transformation techniques into sets of sub steps helps in achieving a tractable implementation complexity. Furthermore, a second benefit results from improved reuse of such sub-steps. The MNEMEE project uses at multiple levels this hierarchical tool composition approach. Most visible is the top-level toolflow. The MNEMEE toolfow is constructed of a configurable set of processing steps, while exposing a regular MACCv2 tool interface to the user. This way, accessing MNEMEE-based memory-aware optimizations in the context of some future project is as easy as invoking a preexisting MACCv2 tool. A closer look at the optimization technique implementation shows the hierarchical tool structure continuing at deeper levels. A particular example is the task graph extraction step. This one consists of three sub steps.

- System modeling: The MACCv2 framework includes means of modeling target-platform system properties. The benefits of using a common target-platform model for each processing step within a toolflow has been presented in the previous section. From the framework perspective, the most beneficial is the unified API for retrieving any kind of system properties.

- ICD-C-based application-code representation: The MACCv2 framework uses the ICD-C-based application-code representation. The major benefit of this code representation is the exact and well-structured representation of the application code as an abstract syntax tree. The MACCv2 framework offers this application-code representation as an annotation to system-model components. The full set of ICD-C code manipulation methods and pre-implemented transformation techniques is exposed to the MNEMEE optimization toolflow. Common tasks, like insertion and modification of statements, can be performed with minimal effort. Sophisticated control and data-flow analysis techniques assist the MNEMEE optimization techniques in performing their tasks.

- Common-object-class model: The common-object-class model is a fundamental service provided by MACCv2 framework. In general, any object class has a common predecessor which provides a rich set of methods for reflection, configuration, storage and construction of dynamic object relations. Please refer to Section 4.2.1 for more details. Especially for the MNEMEE project, the following features of the common-object-class model have been beneficial:
  - Annotations and dynamic object relations: The common-base-class type enables type-safe implementation of object reference collections. These collections are used to implement a key-value map, which stores relations to other objects. In a typical application scenario, annotation of optimization-specific data to items of the system model or application-code representation is performed. MNEMEE uses such annotations to pass optimization results and intermediate data between optimization steps. Especially, the unified base class representation used for both system model and application-code representation has simplified the construction of relations between both domains. A typical example is the representation of mapping decisions, where application-code items are assigned to system-model components.
  - Persistent storage: The common base class provides methods for object-state serialization. Basically, complete object graphs can be stored to disk and reconstructed in subsequent optimization runs. MACCv2 framework uses such persistent object storage for system-model retention, runtime environment and tool configuration. In the MNEMEE context, system-model snapshots are optionally created between each processing step. These snapshots enable analysis of intermediate results, fast reapplication of subsequent steps with different configurations or exploring different mapping options on the same preprocessed application code.
  - Runtime linking: MACCv2 uses operating-system features to dynamically add libraries into a process context. These libraries contain typically system-model components, optimization or transformation steps or interface object definitions for inter-processing-step communication. The general benefit of dynamically loading these libraries is the flexibility to support future object types (i.e. future target-platform models) while maintaining a small process image footprint. Typically, the runtime-linking approach is transparent to the optimization technique developer, due to MACCv2 containing methods for automatic loading of these libraries while performing reconstruction of objects graphs from persistent storage.
  - Object-lifetime monitoring and pointer tracing: The common object class provides methods for tracing of object references and attachment counts. This relieves the developer from implementation effort related to individual garbage collection code. In MNEMEE, this feature saved implementation overhead at several processing steps. One example is the task graph extraction step. Within this step several parallel implementations are maintained for numerous

individual application-code statements. The relation to the original application code is expressed in terms of object annotations. Since these transformations affect code at various nesting levels, keeping track of alive solutions becomes a challenging task. MACCv2-provided object-lifetime monitoring greatly simplified this task.

  - Object-configuration interface: The common base class provides methods for accessing object members in a uniform way. Typically, derived classes need to implement only an enumeration of class members to be exposed via the configuration interface. In MNEMEE, the object configurability has been used throughout the toolflow. Most prominent are the toolflow-level configuration, individual optimization step configurations and configuration of system-model construction.
  - Object factories: Based on the common-base-class model, MACCv2 provides methods for object-class instantiation based on the class name. Factories extend this concept by providing additional control over the object-initialization process. MNEMEE uses this feature primarily for the construction of target-platform system models according to predefined templates. The system-model factories expose a set of configuration parameters to control adjustable properties of the system model. For example, the number of cores or memory sizes can be configured in case of the MPARM system model.

- Runtime environment: The runtime environment as described in Section 4.2.3 enables provision of operating-system-dependent as well as execution-environment-dependent parameters. MNEMEE follows the typical application scenario promoted in MACCv2 by usage of environment variables for resolution of paths to external tools, libraries and temporary files.

- User-interface abstraction: The MACCv2 framework provides methods for abstraction of user interaction. This approach has been described in Section 4.2.4. Basically, the framework provides typical user-interaction methods which, in dependence of current execution environment, will be presented to the user as GUI dialogs or a simple text-mode interface. The optimization or transformation step requesting user interaction does not have to be adapted to use any particular kind of user-interface type. The MNEMEE toolflow uses primarily these features:
  - Dialog: Dialogs are used primarily in the system-model construction tools. There, an interactive mode is available which guides the user though the process of constructing a system model according to a platform template.
  - Progress indication: Since several optimization steps require a prolonged processing time, a detailed progress indication is beneficial to keep the user informed on current actions being performed. MACCv2 supports automatic progress indication at tool level. MNEMEE uses these methods to provide fine-grained progress indication, which includes information on the progress of actions being performed within a step.

- Object views: Object views are MACCv2-provided presentation methods, which enable user interaction beyond simple dialog-based input or option selection. A feasible user-interface implementation (i.e. window-based GUI) can indicate to MACCv2 its capability to display the state of objects of particular type. For example, the graphical UI provided within the MACCv2 framework implements object views for HTML structured text and bitmaps encapsulated as PNG image data. On the other side of the user-interface abstraction, optimization and transformation techniques can request presentation of a particular object in a user-interface-independent way. MNEMEE uses these views to present toolflow runtime statics. Since these object views are not limited to show static data, the taskgraph extraction step can use this user-interaction method to plot its current solution space as a Pareto-graph.

- Platform for dedicated UIs: The MACCv2 user-interface abstraction has been designed to support construction of toolflow-specific user interfaces based on some generic implementation. MNEMEE exploits this feature for construction of a dedicated demonstrator frontend. This is based on the Qt GUI implementation provided in MACCv2, extending this by implementing a toolflow-specific main window which enables easy toolflow configuration as well as progress and result evaluation.

The usage of MACCv2 as a framework for MNEMEE can be concluded as very beneficial for the project. Especially, a significantly reduced implementation effort in combination with improved reuse capabilities, shows the benefit of this approach.

Considering the MNEMEE project structure (which consists of a cooperating set of optimization and transformation techniques) as a typical approach in the domain of memory-aware optimization techniques reveals the versatile applicability of MACCv2 framework services to this domain of such optimization techniques. The framework provides a balanced set of services which covers a wide range of typical tasks occurring in memory-aware optimization techniques. Especially, the common-object-class model provides a pervasive foundation for these tasks, enabling easy data exchange, common notion of object types, simple construction of dynamic object relations and support for persistent data retention. Furthermore, this common-object-class model applies also to the application code and system-model representation. Synergy effects of these unified services result in further implementation-overhead reduction.

Besides implementation-overhead reduction, founding MNEMEE on MACCv2 has positive long term effects. Implementing a toolflow on a modular framework simplifies reuse of MNEMEE optimization techniques. In particular, MACCv2 goes a step further in this discipline. The hierarchical tool representation offers a well-defined interface at various levels. Therefore, either the entire MNEMEE toolflow as a black-box or components of it can be easily reused in further projects.

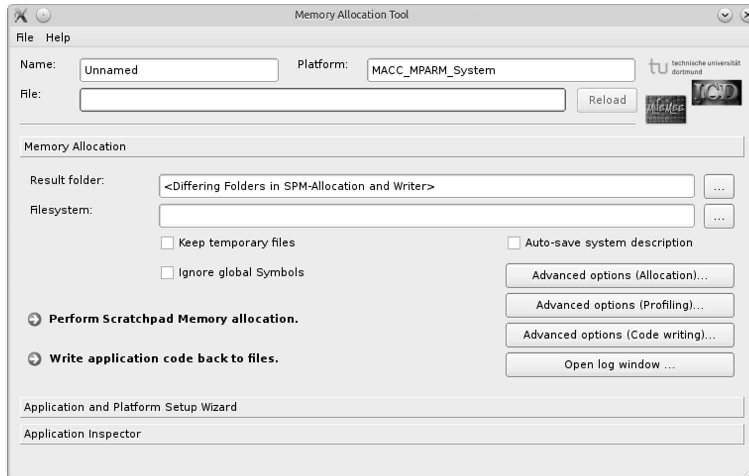


Figure 5.8: Scratchpad allocation tool interface

5.5.2 Application in Teaching

Besides the MNEMEE project, the MACCv2 framework has found usage in teaching activities. Actually, the scratchpad allocation technique, as used in MNEMEE, has been adapted for teaching activities. Marwedel et al. [5] offer courses in the context of ALaRI Master of Advanced Studies Program. The 2010 term included lab excises exploring scratchpad-memory allocation techniques. The MNEMEE-related implementation has been encapsulated into a stand-alone tool. The MACCv2 framework contributed the common services of target-platform description, application-code representation, target-platform compiler interface and user-interface abstraction. Figure 5.8 shows the main interface of this tool-specific GUI.

In particular, the system-modeling approach included in MACCv2 turned out to be beneficial in this application scenario. Since students could tweak target-platform parameters, they were able to conduct various memory allocation experiments in a straightforward way. From the teacher's perspective, preparation of such lab exercises becomes much simpler. MACCv2-provided platform-model templates, tools and applications, which reduced the effort to the core optimization technique implementation. Furthermore, due to the modular structure, the same optimization tool implementation as in MNEMEE could be used. Basically, a significant share of the effort spent on this tool comes from the implementation of a dedicated graphical user-interface frontend.

5.5.3 Application for Present Techniques

The optimization technique evaluated for migration to a MACCv2-based implementation performs operating-system-supported scratchpad allocations. Since being an optimization technique located in the domain of memory-aware optimization techniques for MPSoCs, it is expected to fit well a typical MACCv2 application scenario.

The technique shortly introduced next, has been presented in detail by Pyka at al. [71]. The approach tries to provide scratchpad-memory access in operating-system-equipped environments. This is a challenging task, since typically operating systems do not grant access to physical memory locations as it would be required for common scratchpad-allocation strategies. Therefore, the approach presented in this section adds a scratchpad-memory management component into the operating-system core which controls the scratchpad allocation for each task.

Performing scratchpad allocations solely at runtime, without any further knowledge of application structure resembles typical caching approaches. This approach combines compile-time code analysis and transformations with such a runtime management. This way, the allocation strategy was capable of outperforming hardware caching.

Code analysis is performed at compile-time, which identifies code items, so-called memory objects. Later, access statics are computed for these memory objects. Combining these access statistics of each object with its size gives a benefit value which denotes how worthwhile it would be to place this object into the scratchpad memory at runtime. Access statistics are collected in a profiling step. In a subsequent step, two types of source-code annotations are performed. The first adds data structures which provide memory-object information to the runtime system. Since this approach is expected to work without dedicated hardware support, the second set of transformations introduces an indirection layer which allows assignment of particular memory objects to several memory locations.

At runtime, various allocation strategies have been implemented, starting from simple ones, which place memory objects into scratchpad memory the first time they are accessed. This is followed by approaches, which record memory-object access intervals and mark such object as locked, effectively enabling object displacement of unused objects at context switches. Finally there are approaches which try to use as much as possible of the scratchpad memory for each task, at the expense of higher copy costs, which occur due to saving and restoring of locked object at context switches.

The results show, that a software-based management of scratchpad memory can achieve significant energy savings (up to 83%) even in multi-threaded operating-system-equipped environments. Compile-time annotations and transformations support runtime decisions in a valuable way, effectively enabling this approach to outperform hardware-based caching.

This operating-system-integrated energy-aware SPM allocation technique has been implemented prior to availability of MACCv2. Within this thesis, effort for a hypothetical MACCv2-based implementation is compared to the original one. Since MACCv2 does not affect the runtime environment in both cases, the same operating system and runtime scratchpad manager can be used. Therefore, comparison focuses on compile-time transformations and annotations.

The compile-time transformations are performed in a sequence of steps. In particular these steps can be divided into:

- Identify memory objects to operate on.
- Introduce memory-object access dereferencing.

- Annotate for profiling.
- Compile for profiling.
- Run application and collect profiling data.
- Compute profit values.
- Perform code transformations which introduce memory-object locking.
- Update runtime-data for each memory object .

Implementing this optimization technique based on the MACCv2 framework plus a set of optimization and transformation techniques developed in the context of MNEMEE would reduce implementation effort to:

- Introduce memory-object access dereferencing.
- Compute profit values.
- Perform code transformations which introduce memory-object locking.
- Update runtime-data for each memory object.

Comparing these lists, in particular the tasks of profiling and compiler interfacing could be reused from the MNEMEE project. Since these task are tedious and error prone, an overall effort saving in the development of the compile-time part of this technique is estimated to approximately 60%.

Even though these remaining optimization and transformation steps are focused on this approach, encapsulating them into MACCv2-based processing steps is advantageous in terms of reusability. Most likely the transformation step introducing memory-object locking will find its way into future optimization techniques which propose a combination of compile-time and runtime decisions.

5.6 Conclusion

System-Modeling Evaluation

The contribution of this thesis has been evaluated for fitness and improvement in three dimensions. First of all, the effort necessary for provision of system models has been evaluated. Next, the selection of the abstraction level and finally the precision of this approach have been considered.

With respect to the system-model design effort, detailed implementation effort analysis is performed by estimating the expected number of lines of code. A step by step approach estimates theses values for each component and channel, continuing with the system-level object class and corresponding system-model construction templates. Finally, the effort is added for aspect handlers, which provide typically the information for timing and energy-consumption models. Concluding these results, low initial effort can be observed

for a minimalistic target-platform description. This number of lines of code starts at around 200 lines for a plain host system model. The slightly more complex MPARM model has been defined with an effort of 500 lines, including comments. The effort for adapting system models depends typically on the number of component and channel types. Since for each component and channel class similar effort occurs, the overall effort grows linearly with the number of component and channel classes.

The second evaluation dimension targets the abstraction level. First, the PMS abstraction level has been identified as the most appropriate one. This abstraction level offers a well-balanced ratio between the level of detail needed for memory-aware optimization techniques and the modeling effort. Later on, a mapping between the PMS level description as proposed by Bell et al. and the MACCv2 system model is constructed. The mapping shows straightforward relations between components of both models. Therefore the MACCv2 system description can be considered to be located at PMS level.

Next, the precision of this system-modeling approach is evaluated. The MPARM platform and corresponding system model have been used to demonstrate the achievable level of precision even with limited modeling effort. In particular, energy consumption and runtime have been considered. The actual MPARM full-system simulator provides the baseline. Since the actual optimization techniques and target applications are not at the focus of this evaluation, the results are presented for a well-known scratchpad-memory allocation technique, processing an application code which implements several matrix multiplications. The evaluation targets the key steering parameter of the scratchpad allocation strategy: The gain, in terms of energy consumption, is achievable by moving a particular data item (i.e. a matrix) from its main-memory location into the scratchpad memory. Even though the MPARM system description focuses on the memory-subsystem model, omitting precise bus modeling and peripheral component models, the maximum deviation of MACCv2 calculated gain values compared from the full-system-simulation results is less than 0.21%. This could be improved further once a more sophisticated bus model is implemented into the MACCv2 MPARM system description.

Application in the MNEMEE Project

The MNEMEE project provides a toolchain dedicated towards memory-aware data access optimizations for multicore architectures. This toolchain focuses on the goal of runtime and energy-consumption reduction. This is accomplished by optimized code assignment to processing units, and exploitation of local memory hierarchies for efficient data access. Summarizing the project outcome, a set of optimization and processing steps has been developed. Each of them targets different optimization goals in this toolchain. These goals were access pattern refinement for dynamic data storage, code parallelization and assignment and data item placement in local memories. The MACCv2 framework has been used in this project as the integration foundation throughout the entire toolflow. Well-defined processing-step interfaces, a common-object-class model, data exchange methods, persistent storage of intermediate results and a set of high-level services, as well as the target-platform representation and the ICD-C-based code representation significantly contributed to the positive project outcome. Well-cooperating

tools, reduced implementation effort, easy adaptability to upcoming target platforms and improved reuse of the entire toolflow as well as individual processing steps are only the most noticeable benefits of founding the MNEMEE project on the framework proposed in this thesis.

Application in Teaching and Research Activities

The MACCv2 framework has been used for teaching activities as well. In particular, the scratchpad allocation step developed in MNEMEE has been transfered into a stand-alone code optimization. Students can tweak target-platform parameters and application-code structure to conduct various allocation experiments. MACCv2 in this context provides the target-platform description and the same foundation as for the MNEMEE context. Therefore, implementing these lab exercises was quite straightforward and required only effort for an appealing user interface. The actual allocation technique implementation has been reused without modification.

The last section in this chapter demonstrates as an example the exploitation of the MACCv2 framework for an optimization technique which has been developed prior to availability of MACCv2. The effort for implementing such an operating-system-integrated energy-aware SPM-allocation technique could have been reduced by aprox. 60%, if an implementation would have been performed based on the MACCv2 framework. In particular, the MNEMEE project contributed a set of basic MACCv2 tools which would also be usable for this technique. Such a benefit is mutual in the set of contributing projects. In the case if these techniques had been implemented initially based on MACCv2, the benefit of a reduced effort would have occurred for the subsequent development of MNEMEE. Even though optimization techniques like this one expose a significant runtime-related implementation effort, using MACCv2 as the framework for related compile-time transformations saves this part of the implementation effort and is also beneficial for this type of optimization techniques.

Concluding the results evaluation, this thesis proposes a target-platform description well-suited for memory-aware optimization techniques. It exposes only limited initial modeling effort, while still being capable of delivering precise platform values without the need for full-system simulation. This platform description has been embedded into a framework which has been successfully used in the MNEMEE project and for educational purposes.

6 Conclusion

6.1 Summary

The primary contribution of this thesis is twofold. First, a novel system-modeling approach targeting support for development of architecture-independent source-code optimizations is proposed. Second, a fullfledged optimization technique implementation and integration framework is proposed. Both contributions are tightly related. Together, they significantly alleviate the effort required for implementing source-level memory-aware optimization techniques for MPSoC.

Motivation for the MACCv2 system-modeling approach was the observed demand for a common target-platform description across all parts of a multi-step optimization and transformation approach. In general, common assumptions on the target platform help to avoid optimization decisions which interfere with subsequent steps. In particular, this applies to source-level optimization techniques as well. Even though they operate on an abstract level, knowledge of architectural properties is still beneficial or even vital for this type of optimization technique. The system-modeling approach proposed here provides mechanisms for database-like access to such whole-system target-platform properties, while requiring only definition of locally-scoped input data in terms of component or channel properties. The request-based retrieval of system properties is a unique feature, which makes this approach superior to state-of-the-art table lookup or full-system simulation-based approaches. Table lookups achieve similar response time performance, but suffer from limited applicability scope and difficult update of values, once target-platform modifications occur. Full-system simulations are time consuming and therefore often used only as a postponed evaluation or validation step. Especially, using full-system simulation to guide optimization decisions at optimization technique runtime takes a prohibitively long time.

The approach presented in this thesis has been located at the processor-memory-switch (PMS) level, which provides this target-platform data at the most appropriate abstraction level for source-code optimization techniques. The general structure consists of components representing self-contained entities in the target platform and channels representing the interconnects between them. Items of this system-modeling approach are defined by a unique item class plus a comprehensive set of properties attached to each item. This locally-focused structure enables development of libraries containing sets of components and channels. These libraries are usually grouped according to target platforms being modeled.

Combining these local properties to system-wide-valid data is performed via aspect handlers. These handlers define computational rules which are applied to correlated locally-scoped data along access paths in the memory-subsystem hierarchy. According

to Chapter 5, this approach has been capable of providing close to full-system simulation results. This has been shown for energy-consumption values as well as for access-latency values of the MPARM platform.

The framework introduced in this thesis provides a set of fundamental services to the optimization technique developer. This includes a common-object-class model, a runtime environment and user-interface abstraction. These basic services are used for implementing APIs for system-model representation, the ICD-C-based code representation and processing-step encapsulation. Especially, the processing-step encapsulation and interaction services allow for modular plug-and-play style of toolflow construction. Such processing steps are self-contained entities with well-defined interfaces. The interface definition includes descriptive parameter passing, the actual processing invocation and structured data type definition, usable for system model and application-code annotations.

According to the applicability evaluation in Chapter 5 this framework has been successfully used within the MNEMEE project. The set of optimization techniques provided in this project is organized as a multi-step toolchain. Analysis results are annotated to the target-platform model and to the ICD-C-based application-code representation. The hierarchical processing-step representation in MACCv2 allows for encapsulation of tasks at various granularity levels. This has been frequently used in the MNEMEE project. Starting at top-level the entire toolchain has been represented as a processing-step entity in terms of MACCv2. This allows for embedding the project outcome into future optimization techniques. At the next hierarchy level, top-level project structure is encapsulated into individual processing steps. Complex processing steps are further subdivided into fine-grained code analysis and transformation steps.

Concluding the work presented in this thesis, the system-modeling approach, as well as the framework, show the right set of properties needed to support development of memory-aware optimization techniques. The MNEMEE European Commission funded project, continued research work, teaching activities and PhD theses have been successfully founded on the approaches and the framework proposed in this thesis. The variety of current application fields and the expected continued application of the MACCv2 framework in ongoing research work can be assumed as an indication for MACCv2 being helpful in improving state-of-the-art research work and considered a valuable contribution in the domain of memory-aware optimization techniques for MPSoC platforms.

6.2 Future Work

The MACCv2 framework and system-modeling approach provides a well-defined and feature-balanced environment for development of memory-aware optimization techniques. Nevertheless, some suggestions for further improvement are given next.

The system-modeling approach presented in this thesis provides a full-system model, which is focused on the memory subsystem. Memory hierarchies can be described, including bus bridging and caches. Especially in the case of caches the component modeling may induce higher implementation effort once a very precise cache model is required. The effort comes due to the implementation of aspect handlers which have to take access sequences and scenarios into account. This repeating effort could be reduced once an extended common cache base class and corresponding aspect handlers are provided. Initial in-depth investigation of common cache properties is needed. This includes determining of often occurring cache structures, associativity levels and replacement strategies. Based on these findings, the component class with corresponding set of properties has to be defined. A set of aspect handlers which take these component properties into account has to be implemented as well. Eventually, further derived classes representing major cache types could be useful.

Another possible improvement of the system model targets processing-unit components. In this case, investigation could be performed whether automatic construction of access sequences from application source code is possible. The expected advantage of such an approach would be the possibility to collect aspect values based on application-code structure. A typical example could be abstract estimation of data access energy consumption for a chosen source-code function. Finally, the most promising improvement direction targets a more sophisticated application-code representation model. Currently, the target-application code is attached to a processing unit as ICD-C-based abstract syntax trees. This representation is most suitable if assignment and partitioning in a multicore system is known. In earlier stages where the application code could potentially run on any processing unit, this component-based assignment comes to its limitation. Currently, the system-modeling approach offers the opportunity to assign such an application code to a processing-unit class instead of one particular processing unit. In general, a more sophisticated application-code representation would be beneficial. Especially, once code transformations are performed which split and distribute parts of the application code among the set of available processing units, the links between these subparts may get lost if no dedicated precaution is taken. Therefore, it would be reasonable to add a fullfledged application-code model to the system representation. Besides the yet known ICD-C-based code representation, this could cover topics like shared storage representation, application-code variants management, parallel execution hints and more complex mapping representation.

The system-modeling approach has been embedded into a fullfledged optimization technique development framework. Although the framework provides a rich set of services, some improvement is still possible. Starting from the current processing-step representation, each processing step exposes a single invocable action. Once a set of optimization or analysis steps expose similar functionality or a common parameter set, this may become a limiting factor. In current approach grouping of such co-related processing steps is performed via additional configuration parameters which can be used to select among the set of available processing steps. The downside of such an approach

is that most likely each implementation will use its own way to select between processing options. A preferable approach would be to implement the ability to indicate which processing-step action is requested to be performed in a uniform and framework-provided way. This would result in a well-defined method to invoke and enumerate the set of available processing options in a given MACCv2 tool. As a second example for improved framework services, future development could introduce various system-model iteration methods. Especially, in combination with lambda functions introduced in C++11 [50] this would significantly reduce recurring implementation effort in upcoming optimization techniques.

Optimization and analysis techniques for multicore architectures become increasingly complex. A straightforward approach to cope with the increasing computational effort would be via distribution of tasks onto several computers. The MACCv2 framework provides already a versatile serialization and persistent storage method. Considering this as a foundation for a distributed execution model would be a reasonable next step towards a distributed computing MACCv2 framework. In this context there are numerous interesting challenges to be endeavored including remote tool invocation, parameter passing, application code and system model distribution and synchronization.

Since the MACCv2 framework has shown its applicability in real-life projects like the MNEMEE project, already the by far incomplete set of further improvement directions, indicated here is worth being implemented in the short term. This way, the framework could strengthen even further its contribution to the research community as a tool which simplifies and speeds up development of cutting-edge memory-aware optimization techniques for multicore SoCs.

Bibliography

[1] J. M. Rabaey, A. Chandrakasan, and B. Nikoli, *Digital integrated circuits : a design perspective*. Prentice Hall electronics and VLSI series, Prentice-Hall, 2003.

[2] B. W. Kernighan, *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd ed., 1988.

[3] "ICD-C Compiler framework."
`http://www.icd.de/de/eingebettete-systeme/icd-c-compiler/icd-c`, 2016.

[4] R. Pyka, F. Klein, P. Marwedel, and S. Mamagkakis, "Versatile system-level memory-aware platform description approach for embedded MPSoCs," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, LCTES 2010, Stockholm, Sweden, April 13-15, 2010*, pp. 9–16, 2010.

[5] "Advanced Learning and Research Institute."
`http://www.alari.ch`, 2010.

[6] C. G. Bell and A. Newell, *Computer structures: Readings and examples (McGraw-Hill computer science series)*. McGraw-Hill Pub. Co., 1971.

[7] C. G. Bell and A. Newell, "The PMS and ISP descriptive systems for computer structures," in *In Proceedings of the Spring Joint Computer Conference*, pp. 351–374, AFIPS Press, 1970.

[8] A. Jantsch and H. Tenhunen, eds., *Networks on Chip*. Hingham, MA, USA: Kluwer Academic Publishers, 2003.

[9] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1338–1354, Nov. 2001.

[10] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun, "C Compiler Retargeting Based on Instruction Semantics Models," in *Proc. Design, Automation and Test in Europe*, pp. 1150–1155, 2005.

[11] R. Muhammad, L. Apvrille, and R. Pacalet, "Evaluation of ASIPs Design with LISATek," in *Proceedings of the 8th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '08, (Berlin, Heidelberg), pp. 177–186, Springer-Verlag, 2008.

[12] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC Architecture Description Language and Tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, 2005.

[13] S. Schürmans, G. Onnebrink, R. Leupers, G. Ascheid, and X. Chen, "ESL power estimation using virtual platforms with black box processor models," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pp. 354–359, July 2015.

[14] "PDesigner: Simulator development framework." `http://www.cin.ufpe.br/~pdesigner/`, 2008.

[15] D. D. Gajski, J. Zhu, R. Dmer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Springer US, 2000.

[16] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 2–2, 2007.

[17] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for MPSoC," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 3, pp. 1–18, 2008.

[18] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on*, pp. 29–40, July 2007.

[19] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: a language for architecture exploration through compiler/simulator retargetability," in *Proc. Design Automation and Test in Europe Conference and Exhibition 1999*, pp. 485–490, 9–12 March 1999.

[20] A. Diewald, S. Voss, and S. Barner, "A Lightweight Design Space Exploration and Optimization Language," in *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, (New York, NY, USA), pp. 190–193, ACM, 2016.

[21] D. Kaestner, "TDL: a hardware description language for retargetable postpass optimizations and analyses," in *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, (New York, NY, USA), pp. 18–36, Springer-Verlag New York, Inc., 2003.

[22] C. Kessler, L. Li, A. Atalar, and A. Dobre, "An Extensible Platform Description Language Supporting Retargetable Toolchains and Adaptive Execution," in *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, (New York, NY, USA), pp. 194–196, ACM, 2016.

[23] "IEEE/IEC International Standard - IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," *IEC 62014-4 IEEE Std 1685-2009*, pp. 1–373, March 2015.

[24] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. European Design and Test Conference ED&TC 1995*, pp. 503–507, 6–9 March 1995.

[25] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language For Retargetability," in *Proc. 34th Design Automation Conference*, pp. 299–302, June 9–13, 1997.

[26] P. Mishra and N. Dutt, eds., *Processor description languages*, ch. MIMOLA - A Fully Synthesizable Language, pp. 35–63. Morgan Kaufmann, 2008.

[27] R. Allen, *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[28] J. Magee and J. Kramer, "Dynamic structure in software architectures," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 6, pp. 3–14, 1996.

[29] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[30] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder, "Static Analysis of Energy Consumption for LLVM IR Programs," in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, (New York, NY, USA), pp. 12–21, ACM, 2015.

[31] M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, and L. Benini, "Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations," in *Embedded Computer Systems: Architectures, Modeling, and Simulation, 6th International Workshop, SAMOS 2006, Samos, Greece, July 17-20, 2006, Proceedings*, pp. 279–288, 2006.

[32] Y. Iosifidis, A. Mallik, S. Mamagkakis, E. De Greef, A. Bartzas, D. Soudris, and F. Catthoor, "A Framework for Automatic Parallelization, Static and Dynamic Memory Optimization in MPSoC Platforms," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), pp. 549–554, ACM, 2010.

[33] "Memory maNagEMEnt technology for adaptive and efficient design of Embedded systems."
`http://mnemee.microlab.ntua.gr/`, 2013.

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

[35] D. B. Loveman, "Program Improvement by Source to Source Transformation," in *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, (New York, NY, USA), pp. 140–152, ACM, 1976.

[36] H. Falk and P. Marwedel, *Source code optimization techniques for data flow dominated embedded software.* Kluwer, 2004.

[37] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[38] P. Marwedel, *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems .* Springer Netherlands, 2011.

[39] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and Memory Optimization Techniques for Embedded Systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, pp. 149–206, Apr. 2001.

[40] W. Wolf and M. Kandemir, "Memory system optimization of embedded software," *Proceedings of the IEEE*, vol. 91, pp. 165–182, Jan 2003.

[41] F. Lai, D. Schmidt, and O. Chipara, "Static memory management for efficient mobile sensing applications," in *Embedded Software (EMSOFT), 2015 International Conference on*, pp. 187–196, Oct 2015.

[42] L. Wehmeyer and P. Marwedel, *Fast, Efficient and Predictable Memory Accesses.* Springer, 2006.

[43] M. Verma and P. Marwedel, *Advanced Memory Optimization Techniques for Low-Power Embedded Processors.* Springer, 2007.

[44] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 81–92, Dec 2003.

[45] D. Cordes, P. Marwedel, and A. Mallik, "Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 267–276, ACM, 2010.

[46] O. Jovanovic, N. Kneuper, M. Engel, and P. Marwedel, "ILP-based Memory-Aware Mapping Optimization for MPSoCs," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pp. 413–420, Dec 2012.

[47] D. Cordes, O. Neugebauer, M. Engel, and P. Marwedel, "Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs," in *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, (Washington, DC, USA), pp. 950–959, IEEE Computer Society, 2013.

[48] O. Holzkamp, *Memory-Aware Mapping Strategies for Heterogeneous MPSoC Systems.* PhD thesis, TU Dortmund University, 2017.

[49] D. A. Cordes, *Automatic Parallelization for Embedded Multi-Core Systems using High-Level Cost Models.* PhD thesis, TU Dortmund University, 2013.

[50] "ISO C++ Standard."
https://isocpp.org/std/the-standard, 2016.

[51] "lp_solve API reference."
http://lpsolve.sourceforge.net/, 2016.

[52] "Writing an LLVM Pass." http://llvm.org/docs/WritingAnLLVMPass.html, 2016.

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series, Pearson Education, 1994.

[54] K. Maeda, "Comparative Survey of Object Serialization Techniques and the Programming Supports," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 5, no. 12, pp. 1488 – 1493, 2011.

[55] "Java Interface Serializable."
https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html, 2016.

[56] K. Henney, "C++ Patterns - Reference Accounting," in *in Proceedings of the EuroPLoP 2002 conference, (Irsee*, 2002.

[57] "Java Interface Observer."
http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html, 2016.

[58] "MNEMEE Deliverable list."
http://mnemee.microlab.ntua.gr/public_reports.php, 2013.

[59] "MPARM Project."
http://www-micrel.deis.unibo.it/sitonew/research/mparm.html, 2016.

[60] R. Dömer, *System level modeling and design with the SpecC language.* PhD thesis, TU Dortmund University, 2000.

[61] "Boost C++ libraries."
`http://www.boost.org/`, 2016.

[62] "CoMET - A system engineering tool for the creation of virtual prototypes."
`https://www.synopsys.com/Prototyping/VirtualPrototyping/Pages/CoMET-METeor.aspx`, 2016.

[63] "MSC8144 Quad Core Digital Signal Processor Data Sheet, Rev. 16."
`http://www.nxp.com/files/dsp/doc/data_sheet/MSC8144.pdf`, 2016.

[64] C. Baloukas, L. Papadopoulos, D. Soudris, S. Stuijk, O. Jovanovic, F. Schmoll, D. Cordes, R. Pyka, A. Mallik, S. Mamagkakis, F. Capman, S. Collet, N. Mitas, and D. Kritharidis, "Mapping Embedded Applications on MPSoCs: The MNEMEE Approach," in *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2010, 5-7 July 2010, Lixouri Kefalonia, Greece*, pp. 512–517, 2010.

[65] S. J. E. Wilton and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, May 1996.

[66] "GCC, the GNU Compiler Collection."
`https://gcc.gnu.org/`, 2016.

[67] Z. Hanna and T. Melham, "A Symbolic Execution Framework for Algorithm-Level Modelling," in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009.* (P. Kalla and P. Mishra, eds.), pp. 94–99, IEEE, 2009.

[68] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a MPSoC environment," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, pp. 752–757, 16–20 Feb. 2004.

[69] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, pp. 409–415, 4–8 March 2002.

[70] A. Bona, M. Caldari, V. Zaccaria, and R. Zafalon, "High-Level Power Characterization of the AMBA Bus Interconnect," in *SNUG*, 2004.

[71] R. Pyka, C. Faßbach, M. Verma, H. Falk, and P. Marwedel, "Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications," in *Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pp. 41–50, ACM, 2007.

List of Figures

2.1 System-modeling levels . . . 13
2.2 Exemplary Bell/Newell PMS representation of a CDC 6600 system. [7] . . 14
2.3 ICD-C IR structure overview. . . . 24
2.4 ICD-C workflow. . . . 25
2.5 MNEMEE toolflow overview [33] . . . 28

3.1 System-modeling abstraction levels. . . . 35
3.2 Typical toolflow construction. . . . 37
3.3 Graph structure example. . . . 41
3.4 System-description instance. . . . 43
3.5 Component inheritance tree example. . . . 45
3.6 Channel inheritance tree example. . . . 47
3.7 Address spaces occurrence example. . . . 49
3.8 Access routing example architecture. . . . 52
3.9 Access tree. . . . 53
3.10 Aspect modeling overview. . . . 55
3.11 Aspect-value computation example. . . . 59
3.12 MACCv2 usage scenarios. . . . 61
3.13 Typical optimization steps. . . . 63
3.14 Tool applicability. . . . 65
3.15 Tool-dependency-driven application sequence. . . . 67

4.1 Common object features. . . . 77
4.2 Example of an annotated object graph. . . . 80
4.3 Configuration object structure example. . . . 85
4.4 Object event notification example. . . . 86
4.5 Environment structure . . . 92
4.6 Object-view relations . . . 97
4.7 Logging and progress indication window. . . . 102
4.8 Selection input window. . . . 103
4.9 Eclipse user interface. . . . 104
4.10 System-modeling API. . . . 106
4.11 Host system-model template. . . . 107
4.12 MPARM system-model template. . . . 108
4.13 CoMET system-model template. . . . 109
4.14 MSC8144 system-model template. . . . 111
4.15 Compilation tool hierarchy example. . . . 124
4.16 Tool interface example. . . . 125

4.17 Tool configuration options. 127
4.18 Tool hierarchy. 128

5.1 Initial system model. 133
5.2 Exemplary Bell/Newell PMS representation of a CDC 6600 system [7]. . . 136
5.3 MPARM target platform. 139
5.4 Abstract host architecture 145
5.5 CoMET flat target-platform architecture 146
5.6 CoMET hierarchical target-platform architecture 147
5.7 MSC8144 Quad core DSP architecture [63] 148
5.8 Scratchpad allocation tool interface 155