Memory-based Optimization Techniques for Real-Time Systems

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Sascha Plazar

Dortmund

2012

Tag der mündlichen Prüfung:	19.06.2012
Dekanin:	Prof. Dr. Gabriele Kern-Isberner
Gutachter:	Prof. Dr. Peter Marwedel
	Prof. Dr. Sabine Glesner (TU Berlin)

Acknowledgments

A number of people have in various ways contributed to this work.

In the first instance, I would like to thank my advisor Prof. Dr. Peter Marwedel who gave me the chance to work in his group. He always gave me the freedom and the opportunity for research in many different interesting domains. At the same time, he always lent me his support with his vast knowledge and his advice in numerous discussions. Furthermore, I would like to thank my second referee Prof. Dr. Heiko Falk for his effort and the years we worked together in the same research group. The inspiring cooperation and the countless valuable discussions shaped the content and the style of this work.

At the Department of Computer Science XII, I had the opportunity to work together with great colleagues. I owe special thanks to Dr. Paul Lokuciejewski, Jan C. Kleinsorge and Timon Kelter not only for the numerous technical and non-technical discussions but also for the collaboration on our WCC framework. Furthermore, I have to thank Dr. Michael Engel and Constantin Timm for the extensive technical discussions and proof-reading of this work. Special thanks go to Prof. Dr. Jens Wagner who awakened my interest for the technical aspect of computer engineering and the numerous joint mechanical as well as electronic projects.

In the past years, working with students enriched the teaching part of my work. I am especially obliged to Jens Möllmer, Arthur Pyka and Oliver Jokiel for their cooperation, interesting discussions or having contributed to this work.

The research leading to this thesis has received funding from the European Community's ArtistDesign Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008. Part of the work on this thesis has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3.

Above all, I wouldn't have reached this point without the support of my family which has always been there for me. Special thanks go to my wife Pia which has enriched my life in the past ten years and my parents Wilhelm and Marion as well as my sister Verena. They all gave me love and support during the course of this thesis.

Dortmund, April 2012
Sascha Plazar

Contents

1 Introduction **1**

1.1 Real-Time Systems Design 4
1.2 Compilation for Real-Time Systems 6
1.3 Contributions of this Work 7
1.4 Author's Contribution to this Dissertation 8
1.5 Overview 9

2 WCC – WCET-aware C Compiler **11**

2.1 Related Work 12
2.2 WCET Analysis 13
2.3 Workflow 16
2.4 Components 17
2.4.1 ICD-C Compiler Frontend 17
2.4.2 Code Selector 18
2.4.3 ICD-LLIR Compiler Backend 18
2.4.4 Timing Analysis 20
2.4.5 Flow Facts 22
2.4.6 Memory Hierarchy Specification 25
2.4.7 Available Optimizations 26
2.5 Target Architecture 28

3 WCET-aware Memory-based Optimizations **31**

3.1 Introduction 32
3.2 Existing Code Optimizations 33
3.3 Branch Prediction aware Code Positioning 34
3.3.1 Motivating Example 36
3.3.2 Related Work 37
3.3.3 WCET-driven Code Positioning 38
3.3.4 Evaluation 51
3.4 Cache-aware Memory Content Selection 57
3.4.1 Motivating Example 59
3.4.2 Related Work 61
3.4.3 WCET-driven Memory Content Selection Algorithm 62
3.4.4 Evaluation 65
3.5 Summary 68

4 Optimization of Multi-Task Systems 71
4.1 Introduction . . . 72
4.2 Existing Multi-Task Approaches . . . 73
4.3 Multi-Task Compiler Extensions . . . 74
4.4 WCET-aware Software Based Cache Partitioning for Multi-Task Systems . . . 76
4.4.1 Motivating Example . . . 77
4.4.2 Related Work . . . 78
4.4.3 Software-based Cache Partitioning . . . 79
4.4.4 Size-driven Partition Size Selection . . . 82
4.4.5 WCET-driven Partition Size Selection . . . 83
4.4.6 Evaluation . . . 85
4.5 WCET-driven Multi-Task Program Scratchpad Allocation . . . 88
4.5.1 Motivating Example . . . 89
4.5.2 Related Work . . . 90
4.5.3 Multi-Task Program Scratchpad Allocation . . . 91
4.5.4 Evaluation . . . 93
4.6 Memory Architecture aware Compilation . . . 97
4.6.1 Motivating Example . . . 98
4.6.2 Related Work . . . 99
4.6.3 Optimization Sequence Determination . . . 99
4.6.4 Modification of Optimizations . . . 100
4.6.5 Evaluation . . . 103
4.7 Summary . . . 110

5 Compilation and Optimization for Multiple Targets 113
5.1 Introduction . . . 114
5.2 Related Work . . . 115
5.3 Retargetable WCET-aware Compiler Framework . . . 117
5.4 WCC Extensions . . . 118
5.4.1 Employing GCC for Code Selection . . . 118
5.4.2 General-Purpose Exchange of Information . . . 120
5.4.3 Retargetable Timing Analysis . . . 121
5.5 WCET-aware Static Locking of Instruction Caches . . . 122
5.5.1 Motivating Example . . . 123
5.5.2 Related Work . . . 125
5.5.3 Function-based static I-Cache Locking . . . 126
5.5.4 ILP-based static I-Cache Locking . . . 128
5.5.5 Evaluation . . . 132
5.6 Summary . . . 137

6 Summary and Future Work **139**
6.1 Research Contribution of this Thesis 139
6.2 Future Work 142

A Appendix **145**
A.1 Definitions 145
A.1.1 Basic Block 145
A.1.2 Control Flow Graph 145
A.1.3 Interprocedural Control Flow Graph 145
A.1.4 Knapsack problem 146
A.2 Integer Linear Programming 146
A.2.1 Not Equal Operator 147
A.2.2 Less-than Operator 147
A.2.3 Succeeding Operator 147
A.2.4 AND Operator 148
A.3 Flowchart Symbols 148
A.4 Additional Results 149

List of Figures **154**

List of Tables **155**

List of Algorithms **157**

Bibliography **159**

Chapter 1

Introduction

The history of electrical computing systems is still young [103, 126], nevertheless the speed of development and frequency of innovations has dramatically increased in the past decades: while mainframe systems in the middle of the last century filled whole living rooms or even buildings, their computational performance was very limited. Additionally, these systems have been only accessible by governments, military or research institutes due to exorbitant costs. With the advent of the personal computer in the early eighties of the last century, computing systems have become affordable by everybody. Unprecedented memory sizes were made available by magnetic memories such as floppy disks and hard disks as well as innovations in semiconductor technologies yielding various *random access memory* (RAM) designs. Together with rapidly increasing computational power, personal computers have become popular for various fields of applications like home entertainment, research or in the business sector.

In the beginning of the current millennium, a trend towards smaller computational units which enable mobile application scenarios started. Battery-driven units were integrated into a wider range of products which, so far, were realized purely mechanically with few electrical components. Cars, for instance, were suddenly equipped with a lot of advanced driver assistance systems such as *anti-lock braking systems* (ABS), *electronic stability control* (ESC) or a clutch of airbags. Entertainment and comfort systems (among others *automatic climate controls*, *navigation systems* and *head-up displays*) have been simply impossible before these small and powerful computing systems have been introduced.

Since most of the mentioned computing systems are hidden from the user and are embedded into the actual product, those systems are also called *embedded systems* [82]. In the case of systems heavily interacting with their physical environment such as driver assistance systems or flight control systems, these systems are referred to as *cyber-physical systems* [67]. Along with shrinking in size, embedded/cyber-physical systems became cheaper in production which made them even more attractive for nearly arbitrary purposes. This lead to the fact that practically everybody is surrounded by a wealth of such systems without perceiving them at first sight. Accordingly, this trend is called *ubiquitous computing* [124] or *pervasive computing*.

The following, incomplete list should provide a brief overview of popular domains and application scenarios for embedded/cyber-physical systems:

Automotive: After its invention, automobiles were the domain of mechanical and electrical engineers during the period of a century. Nowadays, the design of automobiles without *electrical control units* (ECU) cannot be imagined: restrictive exhaust emission standards put high requirements on engine control which cannot be realized without the usage of embedded systems. ABS and ESC as active safety instrumented systems indisputably save lives. Therefore, the European Commission regulated their installation in new car models as from 2011 by law [111]. Nevertheless, all such electronic systems can have errors or require adaptions due to new regulations. Since the replacement of such highly integrated components entails large costs, embedded/cyber-physical systems are employed which are usually equipped with rewritable flash memories. This enables easy updates of software applications integrating error fixes, adaption to new regulations or even new features without replacement of hardware. The BMW Series 7 as of 2006, for instance, integrated already up to 70 ECUs with an overall program size of up to 110 MB [90, p. 58–59].

Avionics: In recent times, aircraft became not flyable without electronic help [28, 48, 12]; altitude/heading reference systems and controls are standard and realized by embedded systems. Jumbo jets such as Boeing 747-8 and Airbus A380 are not realizable at all without cyber-physical systems implementing fly-by wire. Technological advances bring comfort to the passengers by intelligent air conditions and individual entertainment systems mounted in back rests.

Handhelds: Handheld devices are usually ultra mobile devices running on batteries. Examples for such devices are first-generation personal CD players which are entirely replaced by small MP3 players. Further examples comprise game consoles, personal digital assistants (PDA), personal navigation assistants (PNA) and smart televisions. Smart phones by now combine the functionalities of mobile phones, digital cameras, PDA, PNA, game consoles as well as audio and video players/recorders. These devices have to cope with a growing amount of multimedia data which steadily increases the demands on embedded/cyber-physical systems. Therefore, powerful embedded systems are developed with dual or even multi-core processors equipped with RAM and flash memories in the range of gigabytes. Such devices already outperform stationary personal computers from the last decade.

Telecommunication: The rapidly growing dissemination of ubiquitous computing does not pass over the domain of telecommunication; the triumph of analog telephone systems was followed by digital communication for speech and data. Various data services such as *digital subscriber line* (DSL), internet via fiber optics or cable connection made fast World Wide Web (WWW) access affordable for everybody. The cordless DECT standard [29] and other radio services such as Bluetooth or WiFi converted stationary installations to comfortable, mobile services. Since

most of these devices are powered 24/7, not only high-performance but also energy conserving embedded systems are required.

As can be summarized from this overview of application scenarios, high performance is one of the most important requirements for embedded/cyber-physical systems. Besides additional, always present demands for low production costs and short time to market cycles, systems employed in ubiquitous or pervasive computing have to fulfill a number of further non-functional criteria [82, p. 5–10] discussed in the following.

Dependability: Embedded/cyber-physical systems have to be dependable which comprises several aspects:

- The probability of failure depends on a system's **reliability**.
- In case of a failure, **maintainability** characterizes time and effort required to bring a system back to normal operation.
- Reliability and maintainability of a system affect its overall **availability** which is the probability that a system is operational.
- In order to avoid damage or even loss of life, high **safety** requirements are imposed to embedded/cyber-physical systems.
- To provide integrity and confidentiality of processed and communicated data, **security** is important for a large number of systems.

Efficiency: Besides demanded dependability, embedded/cyber-physical systems have to be efficient in various ways at the same time:

- Today, almost any electronic system has to be **energy efficient** which is expressed by the buzzword *green computing*. Despite a growing number of embedded/cyber-physical systems, the environmental pollution – and especially the carbon dioxide emissions – should be reduced. In face of a high amount of battery-driven devices, the energy consumption should be reduced as well in order to extend the operating time and save battery cost and weight, respectively.
- Systems should be **runtime efficient** in the fashion of exploiting the underlying hardware platform as effectively as possible. Inefficiency as well as long idle times should be avoided in order to achieve high energy savings and to avoid unnecessarily high clock rates.
- Embedded systems' memories are limited since they are usually not equipped with hard disks. Even though internal flash becomes cheaper, applications stored in it should have a small **code size** to save costs.

- Mobile and handheld devices should have a **low weight** which is often a buying criterion.

- Devices especially in the high-volume segment should have low **costs**. To cut down production costs, as cheap as possible hardware should be used. Therefore, existing hardware resources should be used efficiently in order to avoid an oversizing of the hardware platform. Optimizing compilers and highly specialized applications are crucial to fulfill this requirement [59].

While dependability aspects are mainly influenced by the employed hardware (technology) and software development paradigms, efficiency aspects are in large part depending on the software running on a system. Embedded systems were programmable in assembly at their time of origin. Modern embedded/cyber-physical systems, however, are by far too complex for such a programming style. Growing requirements demand modern development platforms supporting high-level programming languages or graphical design languages. ANSI C [5] was established as de facto standard programming language in the embedded domain and is also synthesized by most of the graphical design tools.

In order to achieve high code quality w. r. t. energy or runtime efficiency, the usage of modern compilers is mandatory which apply aggressive optimizations. In this way, redundant computations can be avoided and specific hardware features can be exploited yielding high software efficiency. Although sophisticated compilers already exploit specialized instruction sets such as *digital signal processing* (DSP) extensions or saturating arithmetic, the programmer is still free to write performance critical parts handcrafted using inline assembly.

Established optimizing compilers – also these ones targeting embedded/cyber-physical systems – almost exclusively focus on *average-case execution time* (ACET) reduction. The latter denotes that runtime that is required in the average case for the execution of a certain program and a representative set of input data. Compilers employ elaborate heuristics for program code and data optimization with the objective of reducing the number of elapsed processor cycles until a program has been processed. Due to the lack of a detailed timing model, a heuristic can only estimate if a modification to apply will actually result in a performance increase. Since particular transformations can result in a performance decrease as well [68], they aim at achieving a runtime reduction added up over all applied code modifications.

1.1 Real-Time Systems Design

Embedded systems are often subject to stringent timing constraints which is why they are often also called *real-time systems*. For such systems, the correctness of a computation does not only depend on the actual computed results but also on the time period within they are delivered. In case of a safety-critical system such as a flight attitude control, exceeding of *hard deadlines* can cause loss of life if resulting

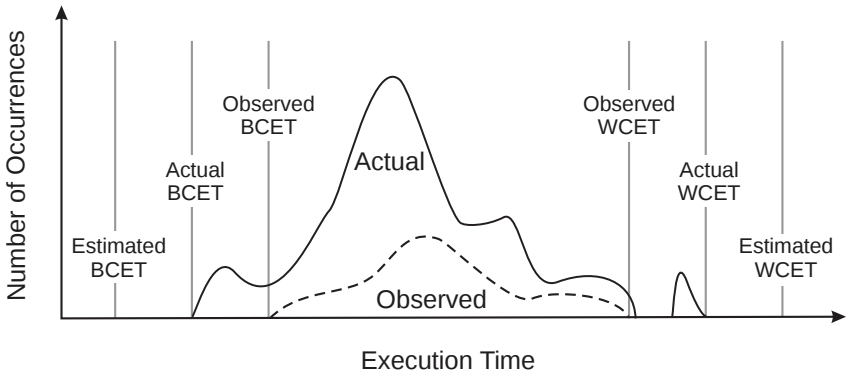


Figure 1.1: Distribution of Execution Times

instable control loops lead to an airplane crash. Thus, the *worst-case execution time* (WCET) of a program or an entire system plays an eminent role in the design phase of such *hard real-time systems*.

Depending on the set of input data, the actual system state and the current environment, a task can exhibit a widespread distribution of execution times. Figure 1.1 depicts an example of such a distribution. The x-axis represents possible execution times for the considered task whereas the y-axis represents the number of occurrences for a certain execution time. According to the choice of the input data set, certain execution times can be *observed* by simply executing the task, denoted by the dashed curve. For common applications running on complex systems, the solution space of possible input data combinations is quite large. Thus, it is exceedingly difficult if not impossible to evaluate all possible input data combinations in order to find the scenario leading to the WCET.

If the worst-case input is not guaranteed to be tested, it can happen that the *actual* WCET, also termed WCET_{real}, exceeds the observed WCET (as shown in Figure 1.1). Hence, measurement based determination of the maximum runtime of an application may lead to an underestimation of the execution time. An unsafe WCET would be the result. Since the WCET is employed for scheduling to verify if tasks meet their deadlines, this not acceptable in a hard real-time environment.

Therefore, static timing analyzers have been developed which allow calculating a program's runtime based on sophisticated analytic models. Static timing analyzers do not execute the program under analysis but analyze its control flow with the help of user-provided annotations in order to determine the *worst-case execution path* (WCEP) triggered by the worst-case input. The WCEP is the longest path inside the *control flow graph* (CFG) of a program, and its length corresponds to the WCET. Due to modern hardware equipped with speculative units such as caches, branch prediction units and possibly speculative execution, it is almost impossible to derive the behavior of all hardware components for every point of time and thereby compute the *actual* WCET. A static timing analyzer thus has to assume

the worst-case behavior in such cases which leads to a certain overestimation. For the *estimated* WCET, also termed WCET_{est}, the following partial order must hold:

$$\text{Safeness}: \; WCET_{est} \geq WCET_{real} \geq WCET_{observed}$$

An overestimated WCET leads to an unwanted – since costly – oversizing of hardware units to guarantee timeliness of the executed tasks. Thus, highly precise static timing analysis techniques are desired to minimize the overestimation:

$$\text{Tightness}: \; WCET_{est} - WCET_{real} \rightsquigarrow min.$$

All techniques and optimizations proposed in this thesis are focused on the reduction of the WCET_{est} which is an important metric of real-time systems. Although the BCET is only infrequently of importance, all approaches proposed in this thesis can be adapted for BCET optimization.

1.2 Compilation for Real-Time Systems

Although the WCET of a system is often a key parameter in the design process of embedded/cyber-physical systems, modern compilers are unfortunately not aware of the worst-case timing behavior. Thus, the design process of embedded application software usually induces a lot of trial-and-error work: A developer has to manually evaluate the WCET of an application employing a timing analysis tool in order to verify if all deadlines are met. If not, the source code of the application has to be optimized by hand and a new WCET evaluation has to be performed. This evaluation/optimization loop is carried out until no more deadlines are violated. Such a software development process is time-consuming and error-prone since a programmer cannot only focus on the actual task – the development of embedded application software.

To overcome this disadvantage and to relieve a developer from repetitive WCET estimations and optimizations, an optimizing compiler which is aware of the WCET as metric is highly desired. Beforehand, the task is to extend a compiler by a detailed WCET timing model. Since static timing analysis is a research domain on its own, an existing, well-tested timing analysis tool should be employed to enable WCET analyses within the compiler. But even if this issue is solved, designing WCET-aware optimizations entails some pitfalls compared to established ACET optimizations: The WCEP of a program is inherently variable, and this frequently requires repetitive, time consuming WCET analyses in order to update the WCET and WCEP information.

Figure 1.2 illustrates the WCEP variability if optimizations are applied. On the left-hand side, an unmodified partial CFG comprising a number of basic blocks is shown (cf. Appendix A.1.2 and A.1.1 for a definition of CFG and basic block). Each basic block has a label and an execution time in processor cycles in brackets. Here, the path `L1`→`L2`→`L3`→`L4` obviously represents the WCEP (denoted by solid

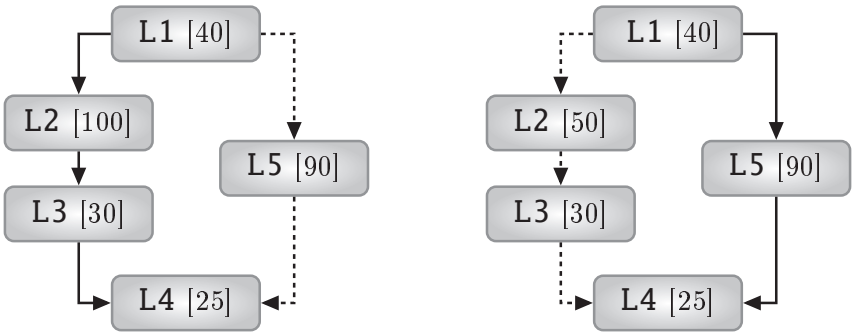


Figure 1.2: Example for WCEP variability

arrows) since its execution time is 195 cycles. The alternative path (denoted by dashed arrows) L1→L5→L4 has a length of 155 cycles and thus does not contribute to the WCET. If, however, an optimization applied to L2 reduces the execution time to 50 cycles, the length of the path through L2 is not any longer the WCET. As shown on the right-hand side of Figure 1.2, the path L1→L5→L4 is the new WCEP and the WCET is reduced to 155 cycles. At this point, further optimizations of L2 or L3 do not reduce the WCET since these blocks are not part of the WCEP.

Hence, optimizations have to keep track of possibly switching WCEPs in order to achieve best possible WCET reduction. The WCET-aware C Compiler *WCC* [34] supports WCET-driven optimizations by integrating the commercially available static timing analyzer *aiT* from AbsInt [1]. In this way, the WCET of a program to be optimized can be automatically evaluated and the current WCEP can be determined. WCC as the platform for the development of WCET-directed optimization techniques therefore serves as basis for the optimizations and extensions presented in this thesis.

1.3 Contributions of this Work

The memory interface of a system often turns out to be a bottleneck which limits the performance of a system. In literature, this effect is also known as *memory wall problem* [84]. In order to reduce the runtime of embedded/cyber-physical systems applications, this thesis proposes numerous optimization techniques which improve memory access times of applications based on their WCET data.

Modern processors try to store instructions likely executed as next in an *instruction fetch buffer* (IFB) which is part of the CPU core. For the first time, the behavior of the branch prediction unit and its impact on the IFB is considered during optimization. Optimization techniques are presented improving the efficiency of this buffer w. r. t. the WCET of a system. If instructions are not found in the IFB, the CPU core performs a fetch from main memory. Instruction caches try to mask these accesses to the main memory by storing local copies in fast small cache memories. In order to improve the efficiency of this part of the memory hierarchy as

well, a memory content selection approach is introduced. The contents of cacheable and non-cacheable memories are determined depending on their influence on the WCET of a program.

Due to growing requirements on embedded/cyber-physical systems, multi-task systems replaced systems running a single application in almost all domains. Established optimizing compilers were not able to keep pace and still lack support for multi-task sets. This thesis presents elaborate extensions for a compiler supporting the compilation and WCET-aware optimization of multi-task systems. This multi-task support was exploited to develop a number of novel optimizations for multi-task systems. As first optimization, a WCET-driven software-based cache partitioning demonstrates the effectiveness of considering the WCET for the optimization of a set of tasks. Furthermore, many embedded systems integrate so-called *scratchpad memories* (SPM) as tightly coupled memories. An optimization approach employing different heuristics is proposed which allows the application of existing single task SPM allocation algorithms in a multi-task scenario. For the first time, a holistic view of memory architecture compilation considers a number of memory-based WCET optimizations and presents approaches for a combined application. An intelligent application order is elaborated and modifications for an interference free collaboration are presented.

Existing compiler frameworks which are able to consider the WCET during optimization are highly specialized and therefore they are limited to a particular hardware platform. In order to support multiple platforms, this thesis presents techniques to ease the integration of standard compilers into a WCET-aware compiler framework. Automatic WCET analyses are supported which allow the application of WCET-driven high-level optimizations to nearly arbitrary target platforms. Feasibility of the presented techniques is shown in this thesis by extending the WCC by a support for the ARM platform. A novel static cache locking optimization demonstrates that assembly level optimizations are supported by this approach. The optimization selects memory blocks which are statically locked into the instruction cache driven by WCET reductions.

All presented techniques and optimizations are implemented as part of the WCC compiler framework. The efficacy of the proposed compiler extensions for multi-task and multi-target compilation is demonstrated by novel optimizations exploiting these extensions. Furthermore, the effectiveness of the presented optimizations is evaluated on a large number of real-world applications.

1.4 Author's Contribution to this Dissertation

According to §10 (2) of "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", a dissertation within the context of doctoral studies has to contain a separate list that reveals the author's contributions to research and results that were obtained in cooperation with others. Thus,

the following list describes the author's contribution to publications which lead to the contents of each chapter:

- **Chapter 3**

  The optimization *cache-aware memory content selection* presented in Section 3.4 was entirely developed by the author. The publication [96] cited in this chapter was written by the author. The other authors contributed by technical discussions and proof-reading of the publication. The idea and the concept of the optimization WCET-driven branch prediction aware code positioning was developed by the author (cf. Section 3.3). The way how the WCET of a program is modeled within an *integer-linear program* (cf. Section 3.3.3.6) stems from Dr. Heiko Falk and his former master's student Jan C. Kleinsorge. Concerning the second publication [93] cited in this chapter, the contribution of the other authors is limited to technical advice and proof-reading. The publication was written by the author of this thesis.

- **Chapter 4**

  The presented multi-task compiler extensions as part of the WCC have been entirely designed and implemented by the author (cf. Section 4.3). Based on these extensions, the optimization discussed in Section 4.4 named WCET-aware software based cache partitioning for multi-task systems was developed by the author of this thesis. The original publication [95] was written by the author and proof-read by the other authors. The memory architecture aware compilation (cf. Section 4.6) was developed by the author in cooperation with his former bachelor student Jens Möllmer.

- **Chapter 5**

  The concept and the implementation of the retargetable WCET-aware compiler framework was developed by the author. The publication [94] cited in this chapter presents early ideas and was written by Dr. Paul Lokuciejewski with contributions by the author. The WCET-aware static locking of instruction caches presented in Section 5.5, which is based on the multi-target extensions, was entirely developed by the author. The publication [92] was written by the author as well. The contribution of the remaining authors was limited to technical discussions and proof-reading.

1.5 Overview

This section provides an overview of the remainder of this thesis:

- Chapter 2 introduces the WCET-aware C Compiler, abbr. WCC, which serves as basis for the approaches and techniques presented in this thesis. WCC's components are discussed with the focus on those parts which are heavily used

and which were extended by the optimizations presented in the following chapters. Techniques for timing analyses are sketched before the tightly coupled static timing analyzer *aiT* and its workflow is introduced.

- In Chapter 3, the memory wall problem is discussed and methods to circumvent it are proposed. Therefore, processor-specific memory-based optimizations are presented which help to mitigate the influence of slow main memories.

- Up to this day, almost all WCET-specific optimizations integrated into a compiler are limited to single tasks. To overcome this gap w. r. t. the design process of embedded/cyber-physical systems, Chapter 4 shows how an optimizing compiler can be extended to consider multiple tasks for WCET optimizations. The effectiveness of this approach is underlined by a two of novel WCET-driven multi-task optimization techniques.

- Chapter 5 tackles the lack of current WCET-aware compilers of supporting multiple targets. Methods are presented which enable easy retargetability of specialized compilers usually restricted to a single target architecture. Again, the applicability of the presented approach is attested by a novel optimization exploiting WCC's multi-task extensions.

- Finally, Chapter 6 concludes this thesis and gives an outlook on possible directions for future research work.

Chapter 2

WCC – WCET-aware C Compiler

Contents

2.1 Related Work . . . 12
2.2 WCET Analysis . . . 13
2.3 Workflow . . . 16
2.4 Components . . . 17
2.4.1 ICD-C Compiler Frontend . . . 17
2.4.2 Code Selector . . . 18
2.4.3 ICD-LLIR Compiler Backend . . . 18
2.4.4 Timing Analysis . . . 20
2.4.4.1 IR Transformation . . . 20
2.4.4.2 Static Timing Analyzer Integration . . . 21
2.4.4.3 Transformation of WCET data . . . 22
2.4.5 Flow Facts . . . 22
2.4.6 Memory Hierarchy Specification . . . 25
2.4.7 Available Optimizations . . . 26
2.5 Target Architecture . . . 28

Embedded/cyber-physical systems are also often (hard) real-time systems which have to satisfy timing constraints. The WCET of such systems thereby plays a significant role during the design process. Reducing the WCET of the software helps to cut down the production cost of a system since possibly slower and cheaper hardware can be used. Thus, optimizing compilers have become popular in the design process of software for embedded/cyber-physical systems which is predominantly written in ANSI C. Even modern compilers lack a detailed timing model: optimizations mainly employ heuristics which apply code modifications based on the assumption that this decreases the average-case runtime of a program. Since the resulting effect cannot be quantified inside the compiler, even adverse effects can be the result. Studies have shown a possible negative effect of code transformations on execution runtimes [18, 17, 30].

Furthermore, the effect of optimizations tailored towards ACET reduction on the WCET of a program is in most cases unknown and almost unpredictable. Optimizations which improve one objective can worsen the other [93, 33]. Due to this uncertainty, embedded system designers are often prevented from using compiler

optimizations for timing critical software. Many applications in the automotive or avionics domain are compiled without optimizations [110] wasting substantial performance.

This chapter introduces the WCET-aware C Compiler *WCC* [34] which is designed for automatic WCET reduction and tightly couples the sophisticated static timing analyzer *aiT* from AbsInt [1]. The compiler is targeted at Infineon's TriCore TC1796/1797 and offers a large variety of standard compiler optimizations. By integrating a fine-grained timing model, WCC allows effective code transformations optimizing the WCET of programs. WCC also serves as framework for the development of dedicated WCET-driven optimization techniques.

The remainder of this chapter is organized as follows: In Section 2.1, a survey of related work is presented which proposes the integration of WCET analyses into compiler frameworks. Timing analysis techniques which allow the estimation of an upper bound of the execution time of programs are presented in Section 2.2. Section 2.3 introduces the workflow of the WCC compiler and presents extensions compared to a standard optimizing compiler which enable a WCET optimizing compilation flow. Section 2.4 provides detailed insight about WCC components which form the basis for the development of WCET-driven optimizations. Finally, Section 2.5 describes WCC's target architecture Infineon TriCore TC1796/1797 with its capabilities and functionalities.

2.1 Related Work

One of the first approaches to integrate a timing analyzer into a compiler framework was carried out by Börjesson in 1996 [11]. The IAR-Systems C compiler for the Intel 8051 processor was extended to read in source code files with user annotations for loop bounds and recursion depths, called *flow facts*. In contrast to WCC's *flow facts* introduced in Section 2.4.5, the *flow facts* used in [11] cannot always be kept valid during compilation since optimizations do not update them. Compared to state-of-the-art microprocessors, the 8051 is fairly simple and predictable since no speculative units as caches, pipelines or branch predictors are integrated.

Another project which tried to couple a compiler with a static analysis tool is *VISTA* [127]. Zhao et al. proposed *VISTA* as an interactive compilation system which assists the developer in finding the best optimization sequence w. r. t. the WCET of a program. In an iterative compilation process where the user can interactively select optimizations to be applied, the WCET of an application to be tuned is automatically recomputed after each modification. Since the compiler lacks a high-level intermediate representation, no source code level optimizations can be applied which wastes optimization potential. The employed proprietary static timing analyzer is rather simple since it features no cache analysis, and the employed loop analysis can handle only simple loops. Since the analysis is based on a path-based approach, it does not scale well and can exclusively analyze small applications.

Heptane, developed by Colin et al. [23], is an open-source static WCET timing analyzer with multi-target support including simple processors such as MIPS or StrongARM 1110. The tool expects C code as input for which a high-level WCET analysis based on a syntax tree can be performed where nodes represent source code structures. Alternatively, the code can be compiled to binary code for which an ILP-based analysis can be employed considering the extracted *control flow graph*. Since the timing analysis relies on a match between the high-level syntax tree and the corresponding low level *control flow graph*, most code transforming optimizations are not allowed. Nevertheless, *Heptane* was used to develop optimizations which keep this mapping valid [49].

The open-source timing analysis tool *Chronos* [70] estimates the WCET of embedded applications targeted at the *SimpleScalar* simulator architecture [6]. A static analysis of binary code is employed which supports complex microarchitectural features such as out-of-order execution, branch prediction as well as data and instruction cache analyses. Chronos is among others utilized for the development of low-level WCET-driven code and data layout optimizations [20] or new static analysis modules for concurrent programs running on shared cache multi-cores [71].

TuBound [97], a compiler framework which is most likely comparable to the WCC presented in this chapter, aims at combining the static WCET analyzer $CalcWCET_{167}$ targeting at Infineon's C167CR and a compiler. *TuBound* processes annotated C code and preserves the supplied information about loop bounds and recursion depths during all applied optimizations. These features resemble WCC's *flow facts* presented in Section 2.4.5. Since a modified GCC version generates assembly code out of the optimized source code, *TuBound* has no support for WCET-driven low-level optimizations.

2.2 WCET Analysis

For applications running on real-time systems with hard timing constraints, their functional correctness does not only depend on the accuracy of computation results but also on the period of time within they can be delivered. In order to verify if timing-critical applications can meet their deadlines, dynamic and static WCET analysis techniques have been developed.

Dynamic timing analyzers, also called measurement-based timing analyzers, determine an upper bound for the execution time of a program by executing or simulating the program and observing the elapsed time. Since the runtime of a program can highly vary depending on its input data, the task is to find the worst-case input data leading to the highest execution time. Due to this uncertainty, a safety margin of, for instance, 25% is added to the highest observed execution time. Nevertheless, it cannot be guaranteed that the WCET derived this way represents a safe upper bound if the selected input data does not lead to the program's worst-case

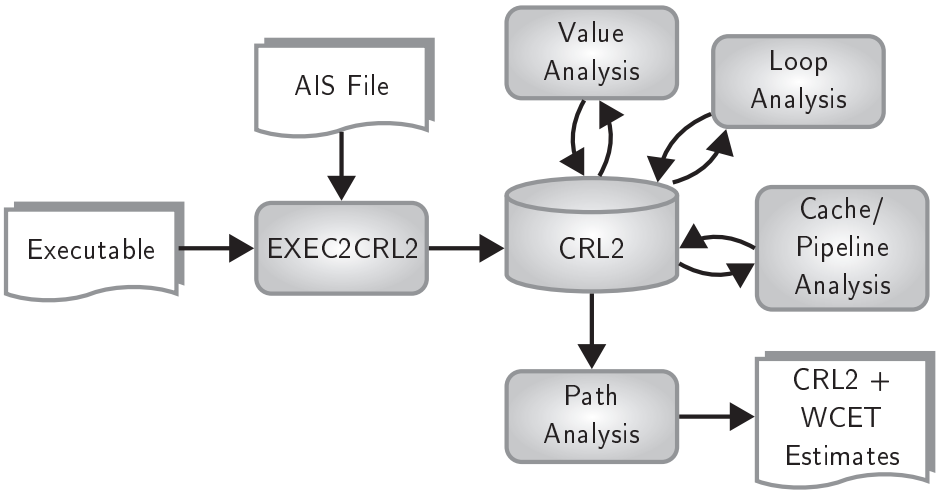


Figure 2.1: aiT WCET Analysis Workflow

behavior [125]. Depending on the program structure and the tested input data, the WCET_{real} can almost arbitrarily exceed the determined WCET_{est}.

Hence, static analysis techniques have been developed which allow a safe estimation of upper bounds of the execution time by applying formal methods. Instead of executing the program under analysis on real hardware, its control flow is statically analyzed to find the set of possible execution paths. The longest path of this set is called *worst-case execution path* (WCEP) and its length corresponds to the program's WCET.

The tool *aiT* from AbsInt is one of the leading static WCET analyzers [1], and it is employed inside the WCC to determine the WCET of a program to optimize. *aiT*'s workflow depicted in Figure 2.1 is described in the following. *aiT* employs a modular system where the static analysis flow is decomposed to separate phases encapsulated in separate programs:

Binary Decoding: Safe estimations of WCETs are only possible if the application is analyzed which will be finally run on the considered platform. Hence, a binary executable is expected as input. The tool `exec2crl` disassembles the binary code in order to reconstruct the *control flow graph* (CFG, cf. Appendix A.1.2) for each contained function. Based on these local CFGs, an *interprocedural control flow graph* (*IPCFG*, cf. Appendix A.1.3) is constructed which is used to explore possible execution paths through the entire program. *aiT* employs its own intermediate representation *CRL2* (*Control Flow Representation Language*) [112] to represent the IPCFG. Important binary properties as memory addresses, variable values or registers are preserved in CRL2.

Although *aiT* integrates a static loop analysis which determines loop bounds of simple loops, such a loop analysis is not decidable in the general case since it includes the proof of the halting problem. Thus, additional information about loop

bounds, recursion depths or address ranges of memory accesses can be specified by the user using *AIS* files. `exec2crl` adds the supplied annotations to the generated CRL2 for processing in the following analysis phases.

The human readable CRL2 representation is passed to all subsequent analysis phases which in turn emit CRL2 as well.

Value Analysis: Register values and the memory content at certain addresses can influence which path of a control flow graph is executed or which memory addresses possibly is read or written. The value analysis thereby tries to determine this data at each program point for every possible *execution context* (different points of time). This can enable an automatic determination of loop bounds, recursion depths and the outcome of conditional branches. In this way, infeasible paths can be classified if the condition of branch instructions always evaluates true or always evaluates false for certain or even all execution contexts.

Loop Analysis: The *loop bound analysis* tries to determine upper bounds for the execution of loops in order to bound their runtime. Therefore a combination of value analysis (cf. Section 2.2) and pattern matching is employed. The analysis success depends on the structure and complexity of the program as well as the employed code generator for which the patterns have to match.

The *loop* and *recursion analysis*, however, exploits the methodology of execution contexts to distinguish between separate loop iterations. When a loop iterates the first time, a number of memory fetches have to be performed in order to fill the cache. Subsequent iterations can profit from the already present cache content probably leading to cache hits. Hence, loop analysis can improve the precision of the WCET analysis and can reduce the overestimation of the WCET.

Cache/Pipeline Analysis: The cache analysis tries to classify each memory access at each possible point of time as a cache hit or a cache miss. Therefore, the results of the value analysis are exploited which can give hints on the target addresses of load/store instructions. Further indications provided by the value analysis can be variable values on which conditional branches can depend. Thereby, the outcome of jump instructions can be derived in order to determine if the outcome results in a cache hit or a cache miss.

The task of pipeline analysis is to determine execution times of each instruction or basic block (cf. Appendix A.1.1 for a definition). Therefore, the behavior of the processor pipelines is modeled. The current pipeline states with their resource occupancies, the content of the instruction prefetch buffer and the classification of memory accesses as cache hits or cache misses have to be taken into account.

Path Analysis: Based on the results of the preceding microarchitectural analyses, the path analysis determines the longest path among all feasible control flow paths. This is done since the length of this path represents the a safe upper bound of the

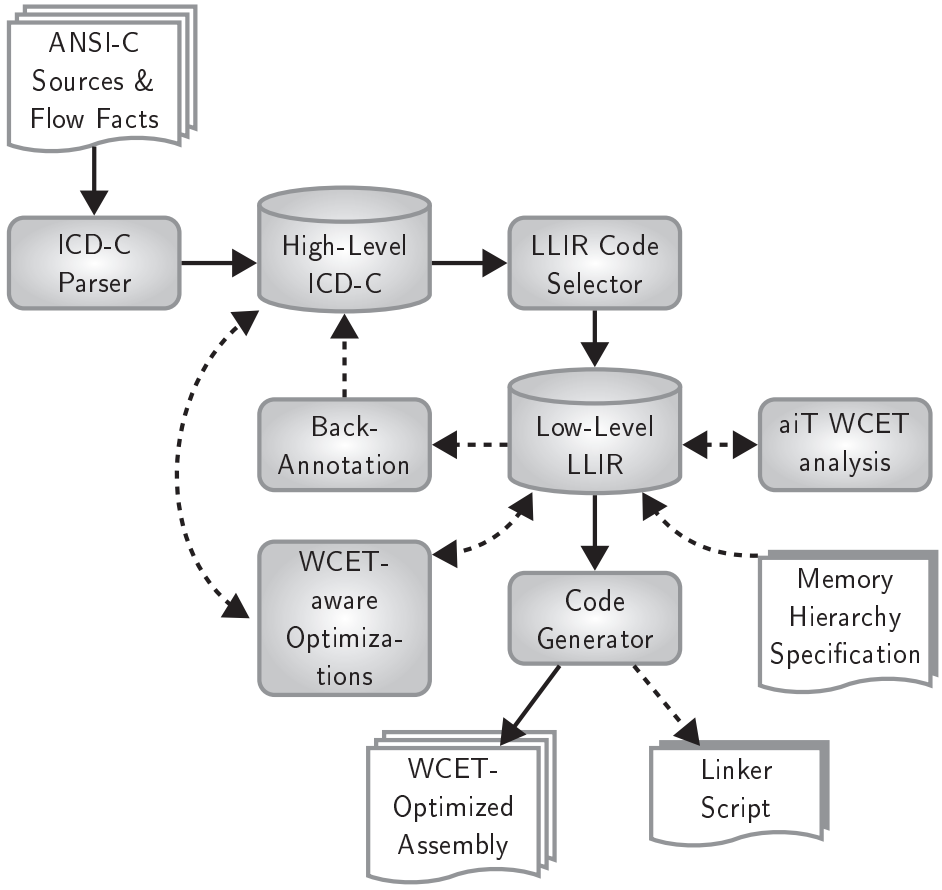


Figure 2.2: Workflow of the WCET-aware C Compiler WCC

execution time – in other words an estimate of the WCET. The program's control flow is therefore modeled using *integer-linear programming* (*ILP*) [8]. The predicted WCET equals the ILP's objective function, and execution counts of control flow edges are represented by the corresponding final ILP variable values.

2.3 Workflow

WCET-driven optimizations and especially the optimizations presented in this work need support of an underlying compiler to collect WCET data and perform required memory layout modifications. The WCET-aware C compiler framework, called WCC [34], assists a developer of high- and low-level WCET-directed optimizations by integrating the static WCET analyzer *aiT* [1].

Figure 2.2 depicts WCC's internal structure (refer to Appendix A.3 for a definition of flowchart symbols). One or more ANSI-C source files of a program with user annotations for loop bounds and recursion depths are read in. Such annotations are called *flow facts* and are required for static timing analysis to bound the execution

time. The source files are parsed by the *ICD-C Parser* and transformed into WCC's high-level intermediate representation (*IR*), called *ICD-C* [100].

At this level, the compiler frontend provides several standard compiler analyses like control- and data-flow analyses as well as various optimizations focussing on ACET and WCET minimization like *Constant Folding* or *WCET-aware Procedure Cloning* [74], respectively.

Afterwards, the component *LLIR Code Selector* translates the high-level IR into a low-level IR called *ICD-LLIR* [27]. All optimizations presented in this these are performed on this TC1796-specific low-level IR which is based on assembly instructions. In order to enable WCET-aware optimizations, *aiT* is employed which performs static WCET analyses on the low-level IR. Therefore, mandatory information about loop bounds and recursion depths is supplied by *flow fact* annotations. These *flow facts* are automatically translated from the high-level IR to the low-level IR and are always kept valid and consistent during each optimization and transformation step of the compiler.

Optimizations exploiting memory hierarchies as presented in this work require detailed information about available memories, their sizes and access times. For this purpose, WCC integrates a detailed memory hierarchy specification available at *ICD-LLIR* level.

WCET data which is provided in the compiler backend and can be exploited by source-level optimizations as well. A module called *Back-Annotation* is employed to translate low-level WCET information back to our intermediate representation *ICD-C* in order to enable high-level WCET-aware optimizations.

Finally, WCC emits WCET-optimized assembly files and generates suitable binaries using an individual linker script reflecting the resulting internal memory layout.

2.4 Components

After Section 2.3 illustrated the overall structure of the WCC framework, some of the individual components should be discussed in detail. The provided knowledge should help the reader to understand how an optimizing compiler operates, how WCET analysis is integrated into the WCC and how WCET-directed optimizations are realized. Based on this description of the components, chapters 3 – 5 indicate how parts of the WCC were modified and extended based on the work presented in this thesis.

2.4.1 ICD-C Compiler Frontend

The high-level intermediate representation (*IR*) *ICD-C* [100] is a target-independent representation of ANSI C source code. It comprises elements such as functions, different kinds of loops or expressions. Sophisticated analyses and a well-defined interface are provided which assist a programmer in the development of even complex source code optimizations. ICD-C itself offers a vast variety of standard compiler

optimizations found in literature such as *constant folding*, *copy propagation* or *function inlining* [88].

2.4.2 Code Selector

Since the ICD-C IR is a target-independent high-level representation of code, a lowering to assembly code is mandatory. Therefore, the TriCore TC17967/1797 specific *code selector* tries to find a semantically equivalent sequence of assembly instructions for each source code construct. An approach based on tree pattern matching is employed to find an optimal sequence w. r. t. a certain objective. A tree grammar augmented with the size of the generated code patterns is used to find an optimal overlapping of the trees with the source code represented by the ICD-C IR. In this way, a code size optimal sequence of assembly instructions can be derived by traversing the trees in the found order.

The code selector also captures information about data accesses within source code expressions and attaches them to the corresponding assembly instructions. Afterwards, low-level optimizations can exploit this information, for instance, in order to move heavily used data objects to faster memories.

2.4.3 ICD-LLIR Compiler Backend

The assembly code emitted by the code selector is represented by the retargetable low-level intermediate representation *ICD-LLIR* [27]. This *low-level intermediate representation* (*LLIR*) is designed for the abstraction of assembly code and therefore provides generic data structures. By extending these generic data structures by a target architecture description, the LLIR is turned into a processor-specific representation.

Equipped with mechanisms and analyses to support the modification, transformation and optimization of the represented assembly code, the LLIR is best suited for the development of assembly level optimizations. Since the code generated by the code selector makes use of an unlimited number of registers, a *virtual* LLIR is generated. By applying an optimization called *register allocation*, the unlimited number of virtual registers is mapped to a limited number of physical processor registers. Hence, the generated LLIR is called *physical* LLIR. The WCC employs a graph-coloring based register allocator [13] by default. On both abstraction levels (virtual, physical), optimizations can be applied which can profit from the selected level of abstraction.

The LLIR's compositional structure is based on the elements which can be found in the assembly sources of a program. A simplified class hierarchy is depicted in Figure 2.3. An object of an `LLIR` class contains the code of a single assembly file which is generated from a single ANSI C source file. For programs consisting of several source files, a list of LLIRs has to be maintained. An LLIR can comprise several `LLIR_DataObjects` representing global variables as found in the C code and `LLIR_Function`s which are named as the corresponding C functions.

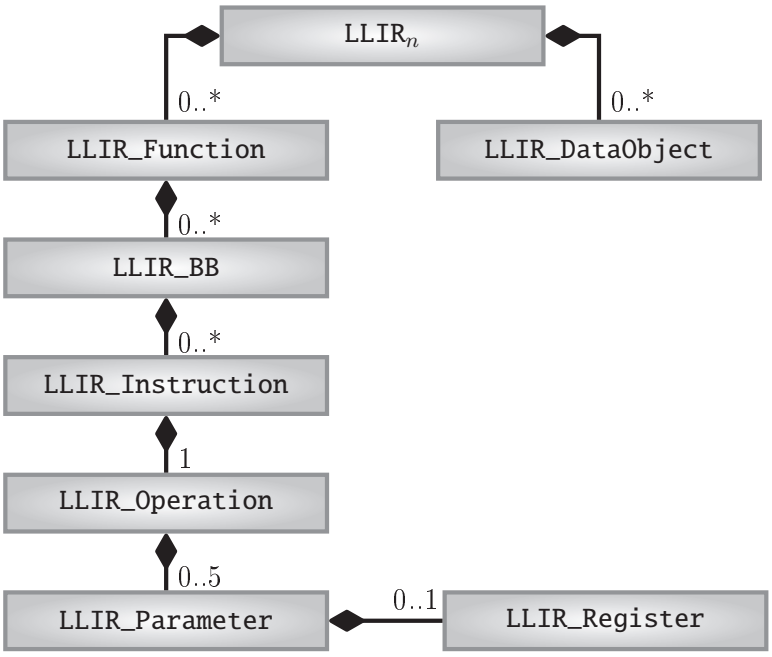


Figure 2.3: ICD-LLIR Class Hierarchy

Each function in turn can be composed of a number of basic blocks represented by the class `LLIR_BB` where a *successor/predecessor* relation specifies the flow of control inside a function. Each basic block can contain a sequence of `LLIR_Instructions`, and each of which can hold one or – in case of a *Very Long Instruction Word* (*VLIW*) processor – multiple `LLIR_Operation`s. Since the TriCore processor does not support explicit parallelization, each instruction consists of a single operation. For each operation, a list of possible parameters can be specified representing constants, labels, instruction set specific operators or virtual and physical registers, respectively. The TriCore instruction set supports up to five parameters for certain instructions. Registers are represented by a dedicated class `LLIR_Register` which provides mechanisms required for register allocation or the specification of hierarchical register sets.

The def/use chains and the register lifetime analysis [88, p. 443–447] make use of the functionality of the `LLIR_Register` class and are examples for the variety of static analysis integrated into the WCC and ICD-LLIR. The TriCore-specific LLIR as compiler backend also offers architecture dependent analyses such as a bit-true data flow analysis which enable processor-specific optimizations [38].

In order to enable feedback-directed optimizations and especially the WCET-driven optimizations presented in this thesis, the LLIR integrates support for detailed objective models. `LLIRObjectives` are generic containers which can keep arbitrary data and can be attached to any LLIR element. The objective handler class `LLIR_Handler` is managing the different objectives attached to a certain LLIR

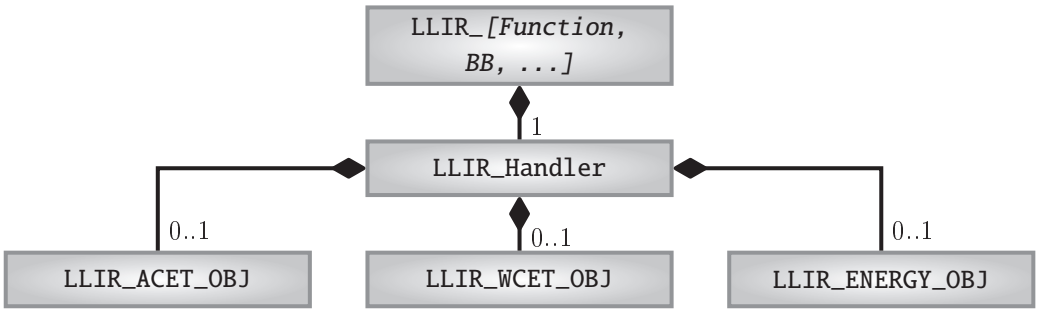


Figure 2.4: Handling of LLIR Objectives

construct. It offers get and set methods for objectives and ensures that only one objective of each type can be assigned to an LLIR element at the same time. The objective which is heavily used for the WCET-directed optimizations presented in this thesis is named `LLIR_WCET_OBJ`. Figure 2.4 depicts the class hierarchy of the LLIR objective handling whereas [75] presents the fundamental concepts of this mechanism.

2.4.4 Timing Analysis

Section 2.2 already presented the workflow of the static timing analyzer *aiT* . The tight integration of *aiT* into WCC's compilation flow is described in the following. *aiT*'s initial workflow (cf. Figure 2.1) requires a binary file as input and is not designed for the output of detailed WCET data. Hence, *aiT*'s tool flow is broken up and a library called *LIBLLIRAIT* encapsulates the invocation of the individual components and was initially published in [35]. Figure 2.5 shows the internal structure of this library which performs the necessary conversion between the different binary intermediate representations LLIR and CRL2 as first step. The generated CRL2 serves as input for the invoked *aiT* timing analyzer in the second step. Finally, the emitted CRL2 is processed to extract the WCET data and convert it to appropriate LLIR WCET objectives (cf. Section 2.4.3) which are attached to the analyzed LLIR.

2.4.4.1 IR Transformation

A reliable timing analysis is only possible on machine code/assembly level where information on hardware timings is available. Hence, *aiT* utilizes its own intermediate representation CRL for representing the binary which should be analyzed. WCC employs a converter called *LLIR2CRL* which replaces the binary decoder `exec2crl` and emits a CRL2 comprising the control flow of the program to analyze. A physical LLIR augmented with WCC's internal memory layout reflecting the symbol table of the corresponding program serves as input.

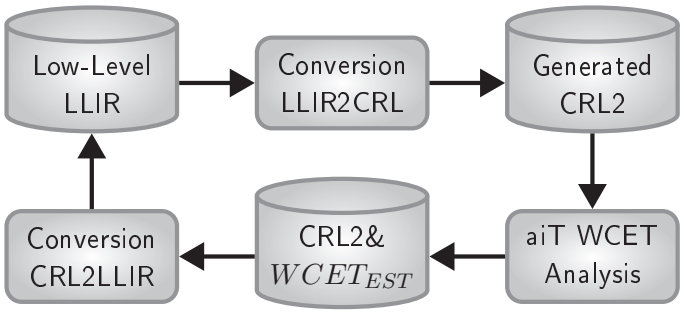


Figure 2.5: Coupling of aiT into WCC

The CRL2 generated by `exec2crl` typically comprises a single CFG which is created by analyzing and transforming a program binary. Thus, the LLIR2CRL converter has to build a global CFG by transforming the structures of all LLIRs of a program as well. Due to the fact that both LLIR and CRL are low-level IRs, the corresponding CRL entity has to be created for each construct such as functions and basic blocks contained in the LLIRs. Besides the latter, memory addresses of all binary structures have to be extracted from WCC's internal memory layout and annotated to each CRL object.

Compared to the creation of CRL functions or CRL basic blocks, the generation of appropriate operations and their parameters is substantially costlier: In the LLIR, operations are uniquely defined by their mnemonics, their bit width – either 16 or 32 bit – and a list of their explicit and implicit parameters. In contrast, CRL operations are identified by a unique integer ID which is created by `exec2crl` when the byte stream of the input binary is processed. Operations with the same mnemonic but different parameter lists or bit widths are mapped to different IDs. Thus, a lookup of the corresponding CRL ID has to be performed for each operation by evaluating all these attributes. Nevertheless, highest accuracy is crucial during the conversion from LLIR to CRL2 in order to avoid deviations of a program represented in CRL2 from the final binary which is produced by assembling and linking the emitted LLIR. Otherwise, guarantees for safe estimations of a program's runtime cannot be made.

In order to allow the extraction of WCET data and assignment to the LLIR after invoking *aiT*, a mapping table keeps track of the LLIR constructs to be analyzed and the corresponding generated CRL constructs.

2.4.4.2 Static Timing Analyzer Integration

The tight coupling of *aiT* into the WCC framework allows an automated evaluation and optimization of a program w. r. t. its WCET. To allow such feedback-directed optimizations which possibly require repetitive WCET analyses, *aiT* has to be invoked without intervention of the user. The WCC therefore controls the application

of the required analysis steps (cf. Section 2.2) and provides the CRL generated by LLIR2CRL as input.

To enable the estimation of runtimes of loops and recursive structures, the initial *aiT* analysis flow requires additional annotations in a user-specified input file. This information is automatically collected by the WCC by evaluating the program's *flow facts* and added to the CRL which serves as input. Details on the cache configuration and available memories with their timings are specified in the same fashion.

This mechanism relieves the developer from the error-prone process of specifying a new *aiT* input file for each analysis if the program structure was modified.

2.4.4.3 Transformation of WCET data

The output of *aiT*'s analysis run is a CRL enriched with detailed timing information. This CRL is processed by the *CRL2LLIR* converter and the timing information is imported into the compiler backend. To establish a relation between the WCET data of CRL constructs and the corresponding LLIR constructs, the mapping table created by LLIR2CRL is exploited. The gathered data is attached in the form of LLIR objectives to the corresponding objects. The following information is extracted and made available within the LLIR:

- worst-case execution times for the entire program, accumulated worst-case execution time for each function and each basic block
- worst-case call frequency for each function
- worst-case execution frequency for each basic block reached by a certain CFG edge
- execution feasibility of CFG edges
- accumulated I-cache misses for each basic block
- approximation of register values

2.4.5 Flow Facts

If a WCET analysis of a program is desired, the maximum iteration count and recursion depths at binary level have to be known in order to determine how often a certain instruction can be executed. By applying static analysis techniques [76], the iteration counts of not too complex loops can be determined, but in the general case, a static loop analysis is not decidable. To enable static WCET analyses yet, the user has to specify those attributes of a program which cannot be determined statically. Besides loop bounds and recursion depths, targets of computed jumps and calls as well as address ranges for memory accesses can be annotated.

Kirner invented the notion of *flow facts* [61] in order to give hints about the dynamic behavior of a program:

Definition 2.1 (*Flow facts*)

Flow facts give hints about possible paths through the control flow graph of a program. Flow facts can be expressed implicitly by the structure of the program itself as also by additional user information.

WCC supports *flow facts* as ANSI C pragmas which are directly inserted into the source code. Such a high-level annotation is a convenient way to specify user *flow facts* since a user usually derives them from the same source code. Additionally, only a single source code base has to be maintained compared to the specification of *flow facts* in separate files. Two different kinds of *flow facts* are supported by WCC's source code annotations: *Loopbounds* and *Flowrestrictions*

Loopbounds: *Loopbounds* are the simple form of *flow facts* and are designed to annotate well-defined loop structures such as `for`-, `while` and `do-while`-loops. They can be employed for regular loops with a single entry point. For such loops with well-defined termination conditions, *loopbounds* describe the minimum and maximum number of loop iterations.

The following *Extended Backus Naur Form* (EBNF) [57] syntax, developed by Schulte [102], describes the specification of *loopbounds* as source code annotation:

LOOPBOUND ⊨ `loopbound min` NUM `max` NUM
NUM ⊨ *Non-negative integer number*

Example 2.1

Assuming that a regular loop has a variable iteration count depending on parameter `cnt` *of function* `foo` *which can be in the range of 20 up to 50, the following code snippet demonstrates the annotation of a corresponding loopbound:*

```
void foo( int cnt ) {
  _Pragma( "loopbound min 20 max 50" )
  while( cnt > 0 ) {
    ...
    cnt--;
  }
}
```

Flowrestrictions: Since *loopbounds* can only be used for regular loops, so-called *flowrestrictions* were introduced. *Flowrestrictions* allow the annotation of irregular loops and recursive function calls by execution ratios of elements related to each other. Therefore, so-called *markers* are employed to identify statements:

MARKER ⊨ `marker` NAME
NAME ⊨ *Identifier*

Function names serve as reference points besides user-defined markers and allow the specification of execution count relations of marked statements employing the following EBNF syntax [102]:

FLOWRESTRICTION	⊨ `flowrestriction` SIDE COMPARATOR SIDE
SIDE	⊨ SIDE + SIDE | NUM * REFERENCE
COMPERATOR	⊨ >= | <= | =
REFERENCE	⊨ NAME | *Function name*

Example 2.2

For a triangular loop, flowrestrictions can be employed to describe the maximum execution count of an inner statement `STMTinner` *relative to an outer statement* `STMTouter`*. Two markers have to be defined in order to identify both statements:*

```
_Pragma( "marker outermarker" )
STMTouter;
for( int i = 0; i < 10; i++ )
  for( int j = i; j < 10; j++ ) {
    _Pragma( "marker innermarker" )
    STMTinner;
    ...
  }
_Pragma( "flowrestriction 1*innermarker <= 55*outermarker" )
```

The inequation which is described by means of the flowrestriction exactly limits the execution count of statement `STMTinner` *to 55 times the execution count of statement* `STMTouter`*. If, however, loopbounds would be used to limit the maximum iteration count of each loop to 10, a maximum execution count of 100 for statement* `STMTinner` *would be the result leading to a potentially overestimated WCET.*

Conversion of flow facts: Up to now, *flow facts* are only available at a very high abstraction level namely the source code level. In order to achieve the goal of a fully automated WCET analysis, the *flow facts* have to be made available at *aiT*'s low-level IR CRL2. Thus, a successive lowering and conversion between all abstraction levels has to be performed.

Figure 2.6 illustrates the necessary conversion steps of *flow facts* starting from the ANSI C source code to the CRL2 serving as input for *aiT*. The parser is responsible for creating and attaching the *flow facts* to the corresponding ICD-C objects. If the code is translated into semantically equivalent assembly code, the code selector also has to lower existing *flow facts*. Attached to the generated LLIR elements, the *flow facts* are evaluated by the LLIR2CRL converter while the input LLIRs

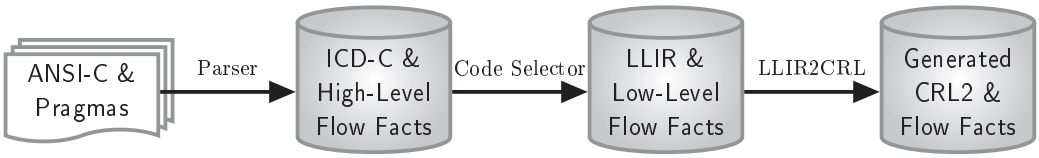


Figure 2.6: Conversion of Flow Facts to CRL2

```
1 [PFLASH-C]
2 origin     = 0x80000000
3 length     = 0x200000   # 2048K size
4 attributes = RXAC       # read/execute/allocate/cached
5 cycles     = 6          # WS; Possible values are 1-7
6 sections   = .text_cached
```

Listing 2.1: Memory Layout Description Example

are processed. Appropriate flow information is created and attached to the emitted CRL2 which can be afterwards analyzed without user intervention.

Optimizations applied on an arbitrary level of abstraction can modify the code in a way that *flow facts* can become invalid. If, for instance, *loop unrolling* is applied, the corresponding *loopbounds* have to be updated and the min/max value has to be divided by the unrolling factor. Even if code blocks are removed or merged with others, the attached *flow facts* have to be kept valid. Markers then have to be moved to a preceding or succeeding block in the control flow graph which is executed whenever the removed block would have been executed in order to keep the related *flowrestrictions* valid. A module called *flow fact updater* provides a uniform interface which is used by all optimizations applied on ICD-C and ICD-LLIR level performing problematic code transformations in order to maintain the set of *flow facts*.

2.4.6 Memory Hierarchy Specification

The execution time of programs does not only depend on the sequence of executed instructions but also on their position in memory. Slower memories cause pipeline stalls, and the processor has to wait until the next instruction has been fetched or data content has been written to or read from the respective memory. Even small memory layout differences can cause substantially differing execution times: if, for instance, instructions or loop headers cross two memory lines, two or more memory lines have to be fetched for their execution. Hence, a WCET-optimizing compiler requires detailed knowledge about the memory layout of a program to optimize or analyze.

WCC therefore employs its own memory layout description comprising all available memories, their sizes and access times. Section assignments can be made either manually or automatically by optimizations which influence the automated generation of linker scripts reflecting WCC's internal memory layout. Listing 2.1 shows a snippet of a memory layout description for the TriCore TC1796 processor. The memory region `PFLASH-C` defined in line 1 is located in the cached memory area at address `0x80000000` and has a size of 2 MB (lines 2–3). The attributes `RXAC` in line 4 indicate that the memory region is readable, its content is executable and should be allocated by the linker. The cached attribute can be evaluated by optimizations, for instance, in order to perform an allocation of heavily used code to cached memory regions. Access times influencing the WCET analysis and section assignments influencing the linker can be specified as well (lines 5–6).

In order to keep track of the memory addresses of arbitrary LLIR constructs, WCC maintains local symbol tables for each LLIR and a global symbol table for the entire program. The symbol tables contain section assignments, sizes and addresses of each code and data object belonging to a program. This mechanism enables the development of approaches performing fine-grained memory layout based optimizations such as the allocation of program code to faster memories as well as the optimizations presented in this thesis.

2.4.7 Available Optimizations

Sophisticated compiler optimizations are mandatory in order to achieve a high code quality. They can be applied at arbitrary abstraction levels such as platform independent optimizations at source code level or target-specific code transformations at assembly level. WCC therefore offers a vast variety of standard compiler optimizations: 23 standard source level optimizations and 11 assembly level optimizations can be applied.

The offered source level optimizations include loop transformations such as *loop unrolling* or *loop deindexing*, data flow optimizations like *constant folding* or *value propagation* as well as the control flow optimizations *function inlining* and *function specialization*. For a detailed discussion of implemented standard compiler optimization, the interested reader is referred to [88, p. 319–704].

The majority of WCC's assembly level optimizations is target specific and tailored to Infineon's TriCore processors. They can be divided into sets which are either applied to a *virtual* LLIR or the *physical* counterpart after register allocation has been performed (cf. Section 2.4.3). Virtual LLIR optimizations are often more powerful since they can make use of an unlimited number of virtual registers which can enable a higher optimization potential. Examples for optimizations applied on a virtual LLIR are *redundant code elimination* and *loop invariant code motion*.

Two optimizations are offered which optimize physical LLIRs. One of them is *instruction tightening* which replaces 32 bit instructions with equivalent 16 bit versions in order to reduce the program's code size. The second optimization is *instruction*

Table 2.1: Available standard Compiler Optimizations

	ICD-C	**Virtual LLIR**	**Physical LLIR**
O0		Register Allocation	Instruction Tightening Jump Correction Silicon Bugs Correction
O1	Constant folding Dead Code Elimination Common Subexpr. Elimination Remove Unused Symbols Simplify Code Value Propagation	Constant Folding Constant Propagation Dead Code Elimination Loop Invariant Code Motion Peephole Optimizations Redundant Code Elimination	
O2	Create Multiple Function Exits Life Range Splitting Loop Collapsing Loop Deindexing Loop Unswitching Merge Identical String Constants Optimize if-Stmts in Loop Nests Redundant Load Elimination Remove Unused Func. Arguments Remove Unused Returns Struct Scalarization Tail Recursion Elimination Transform Head-Controlled-Loops		Local Instr. Scheduling
O3	Function Inlining Function Specialization Loop Unrolling		

scheduling which schedules the list of instructions in a way that TriCore's different pipelines are better utilized in parallel. Different scheduling heuristics thereby help to reduce the overall runtime of a program. Two further code modifications are applied on a physical LLIR but do not directly aim at performance improvements: the *silicon bugs correction* applies workarounds for hardware bugs of the different supported target processors as suggested by the hardware manufacturer [53, 54]. As very last optimization, a fully automated jump correction is applied which corrects the control flow by inserting and removing unconditional jumps as well as negating test conditions. Preceding optimizations which change the order of basic blocks or remove redundant code thus do not need to care about such a jump correction. This simplifies the development of code transformation techniques, saves redundant corrections integrated into various optimizations and ensures that the resulting code will be always be valid w. r. t. its flow of control.

Table 2.1 lists all available optimizations and the abstraction levels at which they are applied. The very left column contains the optimization level for which the optimizations are activated; higher levels include the optimizations applied at lower levels. As can be seen from the table, some optimizations such as *dead code*

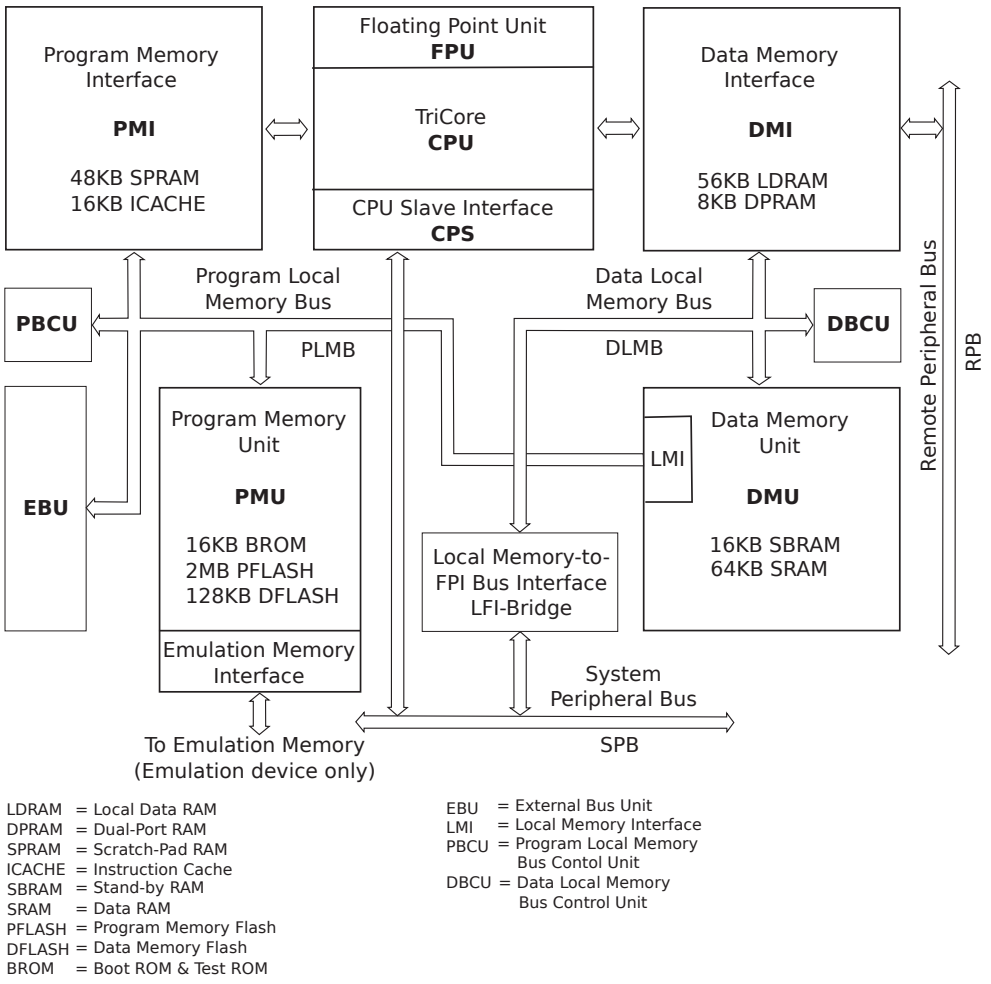


Figure 2.7: Infineon TriCore TC1796 Architecture [52]

elimination or *constant folding* are provided both on WCC's ICD-C and LLIR in order to achieve a high code quality.

2.5 Target Architecture

Since all of the optimizations presented in this thesis are applied at assembly level and exploit target-specific hardware features, this chapter closes with a presentation of the target architecture. WCC targets Infineon's TriCore processor which is designed for the usage in the automotive domain.

The TriCore derivative TC1796 [52] and TC1797 [55] are currently supported. Both are 32 bit RISC microprocessors with Harvard architecture optimized for the usage in embedded systems. Their RISC architecture is extended by a sophisticated

DSP-specific instruction set with multiply-accumulate (MAC) and single instruction multiple data (SIMD) instructions. Saturating arithmetic is available as well as a dedicated floating point unit allowing efficient signal processing. Simplified 16 bit instructions can be used in order to achieve a high code density. Both processors implement a fast hardware controlled context switch logic allowing function calls and task switches in 2-4 cycles.

Three 4-stage pipelines allow triple issues which means that in the best case, three instructions can be executed in parallel. Zero-overhead loop instructions are executed on a dedicated *loop* pipeline, whereas all other instructions are executed either on the *load-store* or on the *integer* pipeline. A static branch prediction unit and an 8 byte prefetch buffer try to fetch the next executed instructions in advance in order to avoid pipeline stalls.

The register file is divided into 16 data and 16 address registers where the upper half (registers 9-16) is automatically saved at a context switch. Moreover, two 32 bit registers of each type can be grouped to an extended 64 bit register in order to express, for instance, the C data type long long by a single register.

Since the optimizations presented in this thesis frequently exploit the memory hierarchy of the underlying system, both supported processors with their memories and buses are discussed separately. Figure 2.7 presents the architecture of the TriCore TC1796 v1.3. Next to the CPU core, SRAM L1 memories for data and program code are located which can be accessed with a latency of one cycle. The so-called *program memory interface* (PMI) is split into an autonomous I-cache and a scratchpad memory for the allocation of content by the user or a sophisticated compiler. *Data memory interface* (DMI), the counterpart on the data side, is split into a data scratchpad and a small dual port RAM. On level 2, the *program memory unit* (PMU) is located which comprises the non-volatile boot ROM, program and data flash. Content located within one of the flashes can be accessed with a latency of six cycles for the first access and two cycles for directly following accesses to the same memory line. The *data memory unit* (DMU) is located on the same level and consists of another data SRAM and a stand-by RAM which keeps its content even if the processor is in deep stand-by.

The internal architecture of the TC1797 v1.3.1 core is depicted in Figure 2.8. Here, again, tightly coupled memories are located next to the core on level 1. Compared to the TC1796, the program scratchpad memory *PMI* is only half as large. The data side, however, comprises a significantly larger data scratchpad and is additionally equipped with a D-cache. Both caches have a configurable size where the unused cache area can be used as extension of the adjacent scratchpads. Both the scratchpad memories and the caches can be accessed with one cycle latency. The *PMU*s on level 2 implement a twice as large program flash but half as large data flash in contrast to the TC1796. The access latencies of six cycles for the first access and two cycles for directly following accesses to the same memory line remain constant.

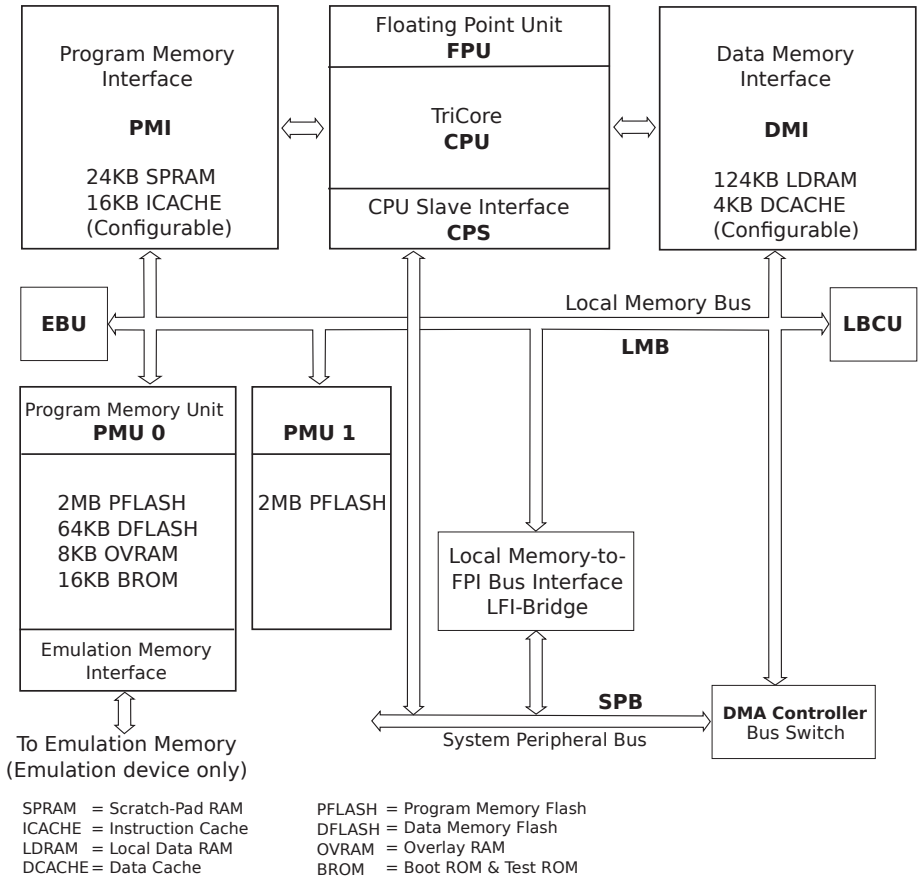


Figure 2.8: Infineon TriCore TC1797 Architecture [55]

Chapter 3

WCET-aware Memory-based Optimizations

Contents

3.1 Introduction . . . 32
3.2 Existing Code Optimizations . . . 33
3.3 Branch Prediction aware Code Positioning . . . 34
3.3.1 Motivating Example . . . 36
3.3.2 Related Work . . . 37
3.3.3 WCET-driven Code Positioning . . . 38
3.3.3.1 Evolutionary Approach . . . 39
3.3.3.2 ILP-based optimization . . . 41
3.3.3.3 ILP Model of the Control Flow of Functions . . . 42
3.3.3.4 Position Constraints . . . 45
3.3.3.5 ILP Model of Jump Penalties . . . 46
3.3.3.6 ILP Model of the Global Control Flow . . . 50
3.3.3.7 Objective Function . . . 50
3.3.4 Evaluation . . . 51
3.3.4.1 Experimental Environment . . . 51
3.3.4.2 WCET Estimations . . . 52
3.3.4.3 ACET Estimations . . . 56
3.3.4.4 Optimization Time . . . 57
3.4 Cache-aware Memory Content Selection . . . 57
3.4.1 Motivating Example . . . 59
3.4.2 Related Work . . . 61
3.4.3 WCET-driven Memory Content Selection Algorithm . . . 62
3.4.4 Evaluation . . . 65
3.4.4.1 WCET Estimations . . . 66
3.4.4.2 Optimization Time . . . 67
3.5 Summary . . . 68

3.1 Introduction

Embedded systems employed in the real-time domain underlie stringent timing constraints in the majority of cases. On the one hand, satisfying such constraints requires a certain amount of computational power. But on the other hand, the hardware platform should be energy-saving and production costs should be minimized simultaneously. In order to fulfill these conflicting goals, advanced compiler optimizations which ensure a high code quality are crucial.

The increasingly growing gap between high processor and low memory performance encouraged the development of memory hierarchies. Frequently used data and code should be allocated to fast but small, tightly coupled memories. This can either be done fully automatically by a *cache* which keeps copies of frequently accessed memory blocks, or explicitly by the user or a compiler exploiting, for instance, so-called *scratchpad memories* (SPM).

Caches grew in popularity since they work transparently from a programmer's point of view which means that no code modification is required. A hardware controller manages the content of the cache which is exchanged automatically with recently used memory blocks. Static analysis techniques have been developed to estimate the cache behavior [39]. However, due to this autonomous behavior, caches are always a source of unpredictability: a timing analyzer tries to determine whether a memory access results in a cache hit or a cache miss in order to consider possible wait states for estimating the WCET. If the outcome of a cache access remains uncertain, the worst case has to be assumed which possibly leads to a highly overestimated WCET.

Scratchpad memories can be accessed as fast as caches since they are manufactured employing the same SRAM technology. Since their content is not automatically exchanged during runtime, scratchpads are fully predictable. The burden of selecting promising content which is allocated to the SPM is shifted to the programmer or an intelligent compiler. Scratchpad memory allocation techniques have been developed aiming for the reduction of a program's WCET_{est} [32, 108].

Besides well-known memories like caches or scratchpads, modern processors employ further memories, which operate in the background, to speed up a program's execution. *Instruction fetch buffers* (IFB) are located inside the CPU core and provide the next couple of instructions to be executed. Branch prediction units determine the outcome of conditional jumps in order to fill the instruction fetch buffer with the correct branch target.

Compiler optimizations which exploit such parts of a system's memory hierarchy require detailed knowledge of the underlying platform. Hardware parameters such as available memories, the bus architecture and resulting access times have to be known as well as attributes of the program to optimize such as the size, address and access frequency of memory objects. Since these parameters are only available at assembly level and usually not accessible at higher levels of abstraction, memory-based optimizations are implemented in a compiler's backend. Hence, all optimizations

presented in this thesis are implemented on ICD-LLIR level and heavily make use of the memory hierarchy specification introduced in Section 2.4.6 on page 25.

This chapter presents new optimization techniques which exploit the memory hierarchy of the TriCore TC1796/TC1797 processor (cf. Section 2.5) in order to decrease the WCET_{est} of a program to optimize. First, Section 3.2 provides an overview of existing code optimization techniques for different types of memories considering different objectives. Then, Section 3.3 presents a new WCET-aware code positioning technique which tries to support the branch prediction unit in order to increase the efficiency of the instruction fetch buffer. An optimized order of basic blocks reduces the number of mispredicted branches and tries to avoid superfluous unconditional jumps. An evolutionary algorithm is employed to fathom the optimization potential of code positioning whereas an ILP-based approach allows acceptable optimization times. In Section 3.4, an optimization aiming at the improvement of the worst-case cache performance is presented. Therefore, the optimization allocates functions which highly profit from a cached execution to cached memory areas. All others are allocated to non-cached memories in order to avoid a mutual eviction from the cache.

3.2 Existing Code Optimizations

Compiler optimizations at assembly level often target average-case runtime reduction and therefore optimize all paths of a program's control flow graph are treated likewise. Even if the performance of certain parts is worsened, the average performance becomes improved in the majority of cases. Standard compiler optimizations such as *constant folding*, *copy propagation* or various *peephole optimizations* are applied in order to remove redundant or useless code and computations [88, p. 329–331, 356–362, 579–605], respectively.

In order to show the implications of code expanding optimizations on instruction cache design, Chen et al. evaluate different types of optimizations and their influence on different cache sizes [21]. Kowarschik et al. give an overview of cache optimization techniques and cache-aware numerical algorithms in [64]. They focus on the memory bottleneck which often limits the performance of numerical algorithms. Both [21] and [64] do not take the impact on the WCET of a system into account.

The influence of the register file size on the average-case performance, on the energy consumption as well on the code size of a program is examined by Wehmeyer et al. in [121]. Employing a parameterizable compiler equipped with an ARM7 energy model, different register file sizes are considered to draw conclusions for each objective.

Steinke et al. employ SPMs to reduce the energy consumption of programs [105]. Therefore, an ILP selects an optimal combination of frequently used program and data objects to be moved in the fast and energy saving SPM. A comparison with the performance of data and instruction caches is also performed. Since the content

of the SPM does not change during a program's runtime, optimization potential is possibly wasted. Therefore, Verma et al. develop dynamic overlay techniques which exchange the content of SPMs during a program's execution [118]. Since content which is no longer used can be replaced be recently used memory blocks, the approach presented in [105] can be significantly outperformed in terms of energy consumption and ACET.

With a growing number of real-time applications, the development of WCET-aware optimizations was brought into focus of research. Since timing analysis of a program is performed at assembly level, most of the optimizations are implemented in a compiler's backend. Redundancy is usually removed by standard compiler optimizations focusing on ACET optimization. Thus, WCET-driven optimizations often aim at reducing the time for which the processor pipeline is stalled, for instance, until a memory access is completed.

Register allocation reduces the time when a pipeline is waiting for accesses to the memory by keeping frequently used data in processor registers. Falk et al. select promising memory objects based on their impact on the WCET [31, 37]. In [31], a standard graph coloring approach is extended by a precise worst-case timing model. Based on this WCET data, spill code is avoided along the WCEP and preferably inserted into concurrent paths. In order to avoid the required reevaluations of the WCET, [37] employs *integer-linear programming* to model the influence of register allocation on the WCEP. The approach extending a code size optimal register allocator [44] outperforms the WCET-aware graph coloring in terms of optimization runtime and achieved WCET reductions.

3.3 Branch Prediction aware Code Positioning

Various code positioning techniques have been developed to improve the processor performance. They have in common that the order of code blocks is modified but exploit quite different hardware features: In systems equipped with caches, code positioning techniques change the order of code blocks to increase the cache performance. Memory blocks which are mapped to the same cache line can cause cache conflict misses and thereby can degrade the performance. The more often such blocks are alternately executed, the more evictions occur. Then, the code is positioned such that blocks which can evict each other are mapped to different cache blocks.

Another code positioning technique which is applicable to all systems aims at reducing the number of executed unconditional jump instructions. Such jumps are, for instance, required for assembly code representing (nested) `if-then-else` statements or loops or combinations of both. Code positioning builds contiguously arranged sequences of frequently executed basic blocks in memory which were formerly connected by unconditional jumps bridging code inbetween. For such basic

blocks where the jump target is the block directly succeeding in memory, the jump instruction can be removed.

The last category of code positioning confines to the optimization of systems equipped with branch prediction units. Branch predictors are developed to work transparently with regard to the software running on a system by integrating a fully autonomous hardware controller. Either dynamic or static techniques are applied for predicting the branch target: dynamic branch prediction units store a history, for instance, by implementing a simple counter denoting whether a branch was taken in the past or not. If the branch has been taken in the past, it also tends to be taken in the future. Thus, the branch target is fetched in advance. Otherwise, the branch was predominantly not taken in the past and the instruction immediately following the branch instruction is fetched from memory. Due to this dynamic behavior, the branch predictor can adapt to changing situations if the outcome of a branch condition changes due to different input data. This often improves the average-case performance but also has the disadvantage that the impact of the branch prediction is hardly predictable by static timing analyzers.

In contrast, a static branch prediction unit determines if a branch will be taken based on static features like the branch direction, the instruction bit width or a dedicated bit in the instruction code. The Infineon TriCore processor TC1796 [52], which is considered in this chapter, predicts 16 bit jumps as always taken whereas 32 bit jumps are predicted depending on the branch direction. Forward jumps (to higher addresses) are predicted not taken while backward jumps are predicted taken. These static features influencing the branch prediction can be evaluated by analyzing the object code without executing the program. Thereby, the influence of the branch prediction can easily be modeled within a timing analyzer.

However, the effectiveness of such a static branch prediction unit highly depends on the control flow of a program and the arrangement of its basic blocks in memory. If the target of a jump instruction can be predicted correctly, the next instruction to be executed can be fetched in advance and thus, the performance can be increased. But if the branch target was mispredicted, the processor pipeline has to be stalled until the next instruction has been fetched from memory. These penalty cycles can lead to a performance decrease. Code positioning therefore rearranges a program's basic blocks such that most frequently executed targets are correctly predicted.

Techniques presented in this section focus on the last two scenarios namely the optimization of unconditional and conditional jump instructions. The remainder of this section is organized as follows: The example in Section 3.3.1 motivates the benefits of compiler-guided code positioning. In Section 3.3.2, an overview of related work is provided. Section 3.3.3 presents the new WCET-driven code positioning algorithms. An evaluation of the performance which is achieved by the WCET-driven branch prediction aware code positioning optimizations is presented in Section 3.3.4.

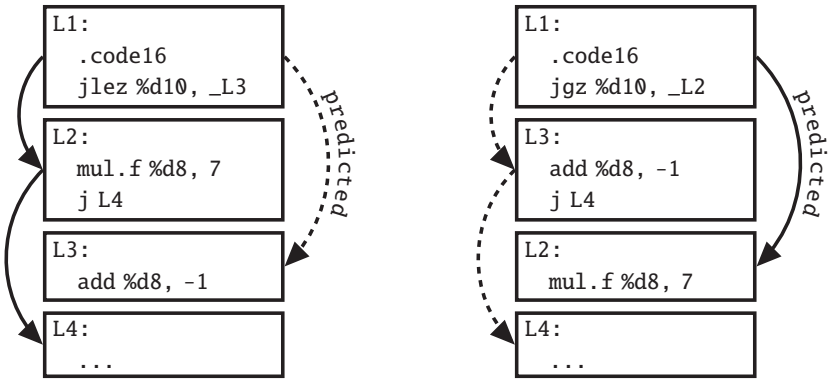


Figure 3.1: Rearranging code layout to support branch prediction

3.3.1 Motivating Example

This section motivates the benefits of code positioning by an example. Branch prediction units fetch the next instruction(s) supposed to be executed from memory in advance. This can either be the instruction which is directly following the branch instruction in memory (*fall-through edge*) or the branch target (*pass-through edge*). The instructions are stored in an instruction fetch buffer which helps to avoid performance-decreasing pipeline stalls where the CPU would otherwise wait for the completion of memory accesses.

If the control flow and the memory layout of a program are ill-arranged, it can happen that the branch targets are predominantly mispredicted by a static branch prediction unit. A high number of mispredicted branches lead to an increased number of pipeline stalls and, as a result, to a performance decrease. Therefore, rearranging the order of basic blocks may be promising in order to support the branch prediction unit.

On the left-hand side of Figure 3.1, a code example for the TC1796 processor and a disadvantageous memory layout is depicted. If register `d10` is less or equal zero, the instruction at the end of `L1` branches to `L3`. Otherwise, the fall-through edge is taken and `L2` is executed which in turn jumps to `L4`. It should be assumed that register `d10` is usually greater than zero, and thus the frequently executed path is `L1` $\rightarrow$ `L2` $\rightarrow$ `L4` represented by solid arrows. Due to the static branch prediction which assumes 16 bit jumps to be taken, the first instruction of `L3` is fetched in advance if the `jlez` instruction leaves the decode stage of the processor pipeline. After evaluating the instruction in the execute stage, the first instruction of `L2` has to be fetched as actual branch target resulting in two clock cycles in which the pipeline is stalled.

Rearranging the code structure as can be seen on the right-hand side of Figure 3.1 helps the branch prediction to fetch the correct instruction in advance. For this purpose, the position of `L2` and `L3` has to be switched and the test condition of `L1`

has to be negated in order to restore the semantics if branching to the new target `L2`. In this way, one processor cycle can be saved for each execution of `L1` since the correctly predicted branch target is fetched advance. As the unconditional jump at the end of `L2` is now superfluous, two additional cycles on the frequently-taken path can be saved; one cycle for executing the jump instruction and one for fetching the branch target. Overall, three clock cycles can be saved. Inserting a jump at the end of `L3` to correct the control flow does not worsen the execution time as long `L3` is not executed.

The *WCC* compiler comprises a jump optimization which *automatically corrects* the control flow by inserting unconditional jumps as well as a correction of branch conditions. Superfluous unconditional branches are removed as well. Thus, its application to optimized programs is not explicitly mentioned in the following.

In order to support an automatic optimization of the code layout which is aware of possible WCEP switches, Section 3.3.3.1 presents an evolutionary approach, whereas Section 3.3.3.2 presents an ILP-based optimization technique.

3.3.2 Related Work

Burguière et al. [14] compare static and dynamic branch prediction in terms of suitability for WCET analysis. They argue for employing static instead of dynamic branch prediction and show that static branch prediction can achieve lower WCETs in most cases. However, Mitra et al. present schemes for estimating the effect of dynamic branch predictors on the WCET of a program in [86]. They derive linear inequations which can be integrated into ILP models for WCET analysis to bound the number of mispredicted branches during execution.

In [42], Gebhard et al. presents a technique for rearranging the positions of tasks to improve the cache performance. The interdependency relation of tasks is evaluated in order to determine a memory layout which maximizes the number of persistent cache sets for each task.

A technique for procedure placement to reduce the cache miss ratio of programs is presented in [45]. Guillon et al. provide an optimal algorithm for memory placement which is improved regarding the sometimes unavoidable code size increase caused by gaps in the address space. In contrast to the optimizations presented in this section, their approach does not target WCET reductions and the order of basic blocks stays untouched which wastes optimization potential.

The authors of [73] present a basic block reordering method based on neural networks. For this purpose, Liu et al. detect typical structures in the *control flow graph* and employ a branch cost model to choose the layout with minimal costs. Unlike the approach presented in this section, their model focuses on the optimization of the average-case execution time and is unaware of the WCET of a program.

Zhao et al. also address the problem of determining improved code layouts which decrease the WCET of a program [128]. As opposed to the optimizations presented in this thesis, only architectures without branch predictors are considered where

unconditional and taken conditional branches always stall the pipeline for a constant number of cycles. An iterative approach is proposed which selects single edges to be contiguous in memory in order to avoid transfers of control.

Bodin et al. aim at improving the WCET of processors supporting compiler-directed branch predictions [10]. By setting a dedicated bit of conditional branch instructions during optimization, the direction to predict is indicated. Optimization potential is wasted, compared to the work presented in this chapter, since unconditional branches cannot be removed due to missing reordering techniques. In contrast to [128] and [10], our approaches are able to optimize both unconditional and statically predicted conditional branches. Both works also do not explore the space of possible solutions in order to evaluate the quality of their results as it is done in this chapter by employing an EA as basis of comparison. Finally, our ILP-based algorithm avoids time consuming repetitive WCET analyses required by state-of-the-art optimization techniques. This is done by explicitly modeling all possible control flow paths as part of the ILP in order to always optimize along the WCEP. Therefore, only *a single* WCET analysis is required.

Falk et al. propose a code positioning optimization [33] aiming at the improvement of the instruction cache behavior. A weighted cache conflict graph based on WCET data is employed for code positioning of basic blocks and entire functions. Based on this formal cache model, two heuristics try to reduce the number of accumulated cache misses and thereby the WCET. Even if code positioning techniques are employed, the impact on the branch prediction is not considered. Thus, synergetic effects of the presented code positioning on the cache performance and the branch prediction cannot be exploited.

Another work considering scratchpad allocation is presented in [107]. Suhendra et al. developed an ILP-based allocation of frequently accessed data objects to faster memories in order to decrease the overall WCET. Their model of the program's WCET and possible execution paths serves as basis for the ILP-based algorithm employed for the technique discussed in Section 3.3.3.2.

3.3.3 WCET-driven Code Positioning

In contrast to existing optimization approaches, the novel branch prediction aware code positioning, which is discussed in the following, is able to optimize both the number of executed unconditional and correctly predicted branches. Therefore, this section presents two novel WCET-driven code positioning algorithms to rearrange the order of basic blocks of a function. The first algorithm employs a genetic approach which starts with a random population. By exploiting the evolutionary techniques crossover and mutation, offspring individuals are generated which desirably converge to the optimal solution w. r. t. the WCET of a program.

Usually, evolutionary strategies can be implemented with small effort and often without understanding the mechanism behind the optimization problem. Even small memory layout modifications, for instance, can have hardly predictable effects on

the instruction fetch unit of a processor. Evolutionary algorithms (*EA*) implicitly consider such effects by evaluating the WCET to determine the fitness values of newly created individuals. Thus, an EA is employed to understand the mechanisms behind the optimization problem and to explore the possible optimization potential of code positioning techniques before developing complex algorithms.

The second optimization discussed in this thesis is a more methodical approach. A more promising order of basic blocks is determined based on an *integer-linear programming* approach. The ILP explicitly models the WCEP as well as the impact on the branch prediction and thereby avoids repetitive WCET analyses.

3.3.3.1 Evolutionary Approach

Finding an improved order of basic blocks inside a function w. r. t. the WCET of a program is a complex task which – in most cases – cannot be done manually. In order to explore the possible space of solutions automatically, an evolutionary approach was developed. In this way, it is possible to fathom the optimization potential of code positioning algorithms with small implementation effort. Such a procedure is advisable before spending time and effort on developing complex optimization techniques with uncertain practical effect.

Evolutionary algorithms stem from the domain of artificial intelligence and implement the principles of biological evolution. By employing reproduction mechanisms including mutation and recombination, offspring generations are created. From a new generation, stronger individuals w. r. t. a certain fitness function are selected as parents for the next generation. With such an approach, preferably improved solutions are "cultured" instead of tackling an optimization problem analytically.

The PISA framework [9] was employed which defines a common interface for the communication of the so-called *selector* and *variator* modules (cf. Figure 3.2). The selector is the algorithm which is responsible for picking out individuals for the *archive* containing promising individuals for later use and as parents for an offspring generation, whereas the variator models the application scenario and implements the problem representation such as the code positioning or an approach for solving arbitrary problems. The variator is responsible for creating offspring individuals from parents by applying the mentioned crossover and mutation operators as well as for the evaluation of the fitness value. Parent and offspring individuals are maintained in a pool called *population*. There are several optimization algorithms for single- and multi-objective optimizations problems providing an interface for PISA. The *Strength Pareto Evolutionary Algorithm 2* (SPEA-2) [129] shows good performance for different numbers of objectives at negligible computational power requirements. Hence, it was chosen as selector although a simpler algorithm could be applied as well.

The presented implementation of the variator creates individuals which represent the order of basic blocks by a mapping of consecutively numbered positions

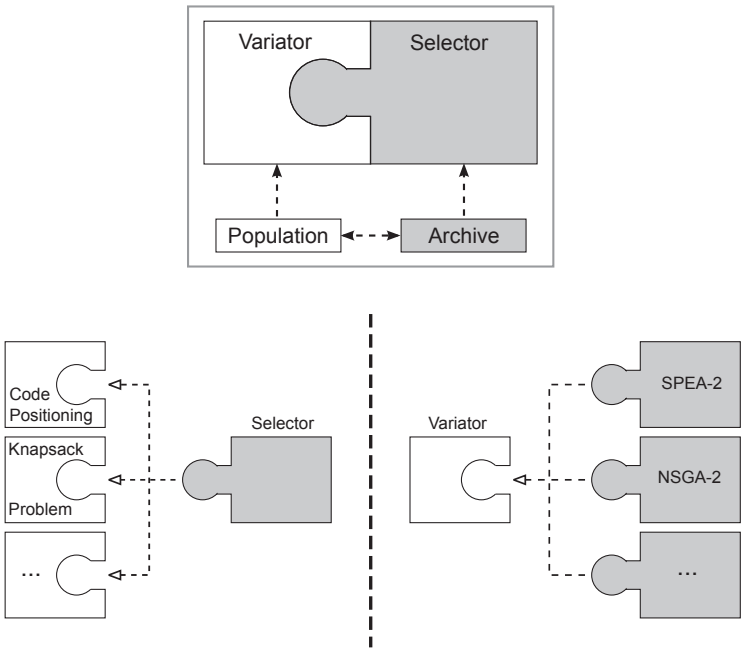


Figure 3.2: PISA Framework

in memory to basic block names. Figure 3.3 illustrates the representation of the basic block mapping of individuals: for each function $f_1 \dots f_n$, a vector stores the basic block names, and the position of the blocks inside the vector correspond to the order of blocks in memory and thereby encodes their position (e.g. position $1 \dots i$ for function f_1 consisting of i basic blocks).

The most commonly used one-point crossover is applied at a random position i as recombination operator to create new individuals. As demonstrated in Figure 3.4, the first i of n functions (here $i = n - 1$) of the first parent individual are combined with the last $n - i$ functions of the second parent individual to create an offspring individual. Crossover points have to be aligned at function boundaries to guarantee the creation of valid individuals. Otherwise, an invalid individual could be the result if, for instance, crossover is applied in the middle of two genes {`L1,L2,L3,L4`} and {`L1,L3,L2,L4`} representing the same function with different orders of basic blocks. The resulting genes {`L1,L2,L2,L4`} and {`L1,L3,L3,L4`} exhibit duplicate as well as missing basic blocks leading to a semantically different behavior.

A newly created one is mutated with a probability of 1 by exchanging two randomly chosen basic blocks of a randomly chosen function. New individuals could be generated by applying only mutation to a single parent individual (without crossover) as well. But this would reduce the speed of exploring the solutions space since good characteristics of two parent individuals could not be combined.

For the evaluation of the fitness of an individual which corresponds to the WCET of the modified program, a WCET analysis employing *aiT* is performed. Since

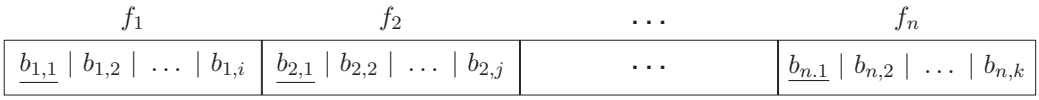


Figure 3.3: Encoding of basic block positions.

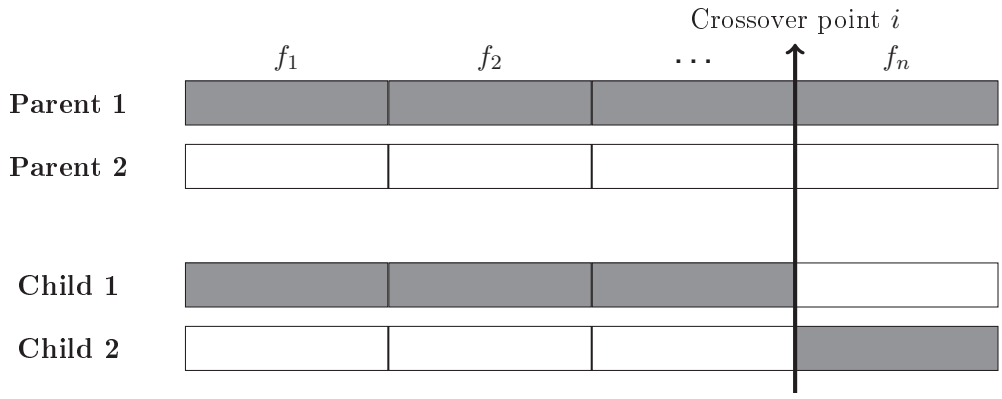


Figure 3.4: One-point crossover reproduction

potentially many individuals are created, a high number of time-consuming analyses is required. A method to avoid redundant WCET analyses is a lookup table for already evaluated solution vectors: if an older individual represents the same solution vector as a newly created one, the appropriate WCET is read from the lookup table instead of invoking *aiT*.

A more sophisticated way to determine an improved order of basic block positions without the need of repetitive WCET estimations is explained in the following section.

3.3.3.2 ILP-based optimization

Trial-and-error approaches realized by evolutionary techniques as presented in the last section are often time-consuming. This is due to the repetitive creation of individuals and evaluation of the fitness function. In contrast, an analytic strategy may yield a problem-aware optimization technique which also leads to good or even optimal solutions. Often, the disadvantage of such "methodological techniques" is their complexity. They also require a high level of knowledge w. r. t. the optimization problem on the part of the developer. Especially for WCET-driven optimizations, the recognition and handling of possible WCEP switches makes optimizations challenging.

This section presents the novel ILP-based optimization technique which is capable to model a program's control flow in order to always optimize along the WCEP.

It determines an improved order of basic blocks w. r. t. the WCET of a program. The algorithm requires only *a single* WCET analysis and is able to consider the influence of the code layout on the branch prediction.

Section 3.3.3.3 describes the modeling of a function's control-flow in the ILP whereas Section 3.3.3.4 introduces constraints steering the position of basic blocks. In Section 3.3.3.5, jump penalties for various jump scenarios are modeled. Finally, Section 3.3.3.6 models the global control flow whereas Section 3.3.3.7 describes the ILP's objective function.

3.3.3.3 ILP Model of the Control Flow of Functions

By applying optimizations, the worst-case execution path can possibly switch. In order to keep track of this critical path, the control flow within functions and the paths' lengths representing their execution times has to be modeled within the ILP. In the following, ILP variables are represented using lowercase letters whereas constants are represented by uppercase letters. The costs C_i of basic block b_i represent the WCET of this block for a single execution as part of the unoptimized program.

Definition 3.1 (*Reducible Control Flow Graph*)

> According to [88], a *control flow graph* $G = (V, E, s)$ is reducible iff. E can be partitioned into disjoint sets E_F, the set of forward edges, and E_B, the set of backward edges, such that (V, E_F, s_F) forms a *directed acyclic graph* (DAG) in which each node can be reached from the entry node S_F. All edges in E_B are back edges which connect nodes with one of their ancestors.

According to Definition 3.1, subgraphs of a reducible *control flow graph* can be collapsed into into single nodes by applying transformations. Such a collapsing can be successively applied from inner subgraphs (usually starting at innermost loops) to outer subgraphs such that the *CFG* can be turned into simpler graphs. Ultimately, the entire graph can be reduced to a single node.

For such reducible *CFG*s, an innermost loop L of a function F has exactly one basic block b^L_{entry} being the loop's unique entry point, and possibly several back-edges turning it into a cyclic graph. Not considering these back-edges turns L's CFG into an acyclic graph. $G_L = (V, E)$ denotes this acyclic graph in the following. It can be assumed that there is at least one basic block b^L_{exit} in G_L being the loop's exit node. The WCET w^L_{exit} of block b^L_{exit} is equal to its costs:

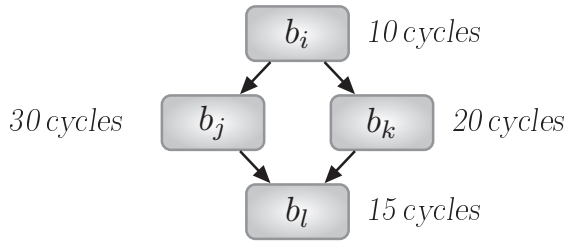
$$w^L_{exit} = C^L_{exit} \tag{3.1}$$

The WCET of a path leading from a node $b_i \neq b^L_{exit}$ of G_L to one of the exit nodes b^L_{exit} must be greater than or equal to the WCET of any successor of b_i in G_L, plus the cost C_i of b_i:

$$\forall b_i \in V \setminus \{b^L_{exit}\} : \ \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + C_i \tag{3.2}$$

Example 3.1 (*Path Modeling*)

In the following, a small example illustrates the modeling of the WCEP for a sequence of basic blocks:



The annotated execution time for each block b represents its costs C. A path is modeled bottom-up starting at basic block b_l. As per Equation (3.1), its WCET only depends on the costs C_l – the time for a single executions of b_l:

$$w_l = C_l = 15$$

According to Equation (3.2), the WCET of block b_j (b_k is modeled analogously) is equal to its own costs plus the WCET of its only successor b_l:

$$\begin{aligned} w_j &= C_j + w_l = 30 + 15 = 45 \\ w_k &= C_k + w_l = 20 + 15 = 35 \end{aligned}$$

Finally, the WCET of the first basic block b_i which ends with a conditional branch instruction is modeled as follows:

$$\begin{aligned} w_i &\geq C_i + w_j = 10 + 45 = 55 \\ w_i &\geq C_i + w_k = 10 + 35 = 45 \end{aligned}$$

Based on the costs and the resulting WCETs, the path $b_i \rightarrow b_j \rightarrow b_k$ is the WCEP since it is the longest path with 55 clock cycles representing the WCET of the whole sequence of basic blocks.

During optimization, a WCEP switch of a program can only happen at such points in the CFG where a basic block b_i has more than one successor. Only there, forks in the control flow are possible where the outgoing paths can have different WCETs. But since Equation (3.2) is formulated for each successor of b_i, variable w_i always reflects the WCET of any path starting at b_i – irrespective of the fact which successors are actually part of the current WCEP. This way, the constraint of Equation (3.2) realizes the implicit consideration of WCEPs and their changes in the ILP.

Since paths are built bottom-up, variable w^L_{entry} models the WCET of all paths of a loop L if it is executed exactly once. In order to model multiple executions of L, all CFG nodes $v \in V$ of G_L are represented by a super-node v_L. The costs of v_L

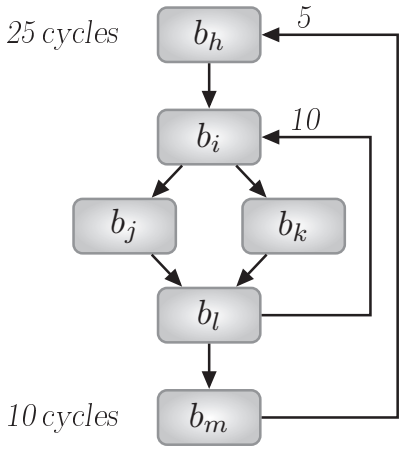
are the product of L's WCET for a single execution and L's maximal loop iteration count:

$$c_L = w^L_{entry} * Count^L_{max} \tag{3.3}$$

Replacing a loop L by a super-node v_L in the CFG may turn another loop L' of F directly surrounding L into an innermost loop with acyclic CFG G'_L. Hence, the constraints of Equations (3.1) to (3.3) can be formulated for L'. This way, the innermost loops of F are successively collapsed in the CFG so that ILP constraints modeling F's control flow are created from the innermost to the outermost loops.

Example 3.2 (*Loop Representation*)

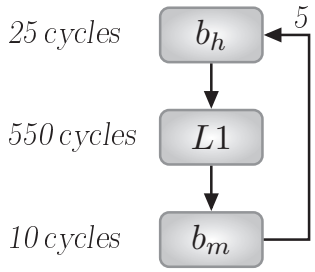
If the sequence of basic blocks in example 3.1 is turned into a loop named $L1$ which is an inner loop of a loop named $L2$, the resulting control flow can be as follows:



According to Equation (3.3), the costs of the supernode v_{L1} representing loop $L1$ amount to the WCET of its entry basic block b_i multiplied by its maximum iteration count:

$$w_{L1} \;=\; w_i * Count^{L1}_{max} \;=\; 55 * 10 \;=\; 550$$

Thereby, the subgraph of the innermost loop $L1$ can be reduced to a node $L1$ turning the outer loop $L2$ in an innermost loop:



Now, the modeling of the path $b_h \rightarrow L1 \rightarrow b_m$ is done as in Example 3.1 before the loop $L2$ is reduced just as $L1$ in this example. The WCET of $L2$ amounts to:

$$w_{L2} \;=\; w_h * Count_{max}^{L2} \;=\; (25 + 550 + 10) * 5 \;=\; 2925$$

Of course, the WCET w_i of 550 clock cycles only hold for the concrete values of this example as long a no path switch in Example 3.1 occurs. But due to the nature of Equation (3.2), the equations in this example always consider the actual WCET w_i and thereby model the WCEP.

The fundamental structure of the ILP constraints of Equations (3.1) – (3.3) stem from the work of Suhendra et al. proposed in [107]. In order to implement a fully functional code positioning technique, these basic constraints had to be refined substantially. Extensions of the original ILP formulation are described in the following sections.

3.3.3.4 Position Constraints

For a function F, the order of its basic blocks in memory is consecutively numbered. The position of a basic block b_i inside F is represented by an integer variable x_i as part of the ILP model. Thus, the value of x_i represents the absolute position inside F which consists of N basic blocks:

$$x_i \in \{1, ..., N\} \tag{3.4}$$

The decision variable x_i for a basic block b_i allows a free positioning of each basic block inside a function. However, there are some constraints which have to be taken into account. First, without loss of generality, each function F has one dedicated entry block b_{entry}^F with its corresponding decision variable x_{entry}^F which has to be kept as first block of the function:

$$x_{entry}^F = 1 \tag{3.5}$$

Furthermore, at each logical position of a function F, exactly one block must be assigned. Let V be the set of F's basic blocks, b_i and b_j two basic blocks with their corresponding decision variables x_i and x_j, respectively. Then, a number of constraints have to be formulated to ensure that there are no two variables with the same value in order to avoid several basic blocks at the same position:

$$\forall b_i, b_j \in V : b_i \neq b_j \Rightarrow x_i \neq x_j \tag{3.6}$$

Due to the value range of x_i, defined in Equation (3.4), Equation (3.6) implicitly ensures that each basic block is allocated to a valid position. For the formulation of the $\neq$ operator, please refer to Equation (A.1) in the Appendix.

Table 3.1: TriCore Jump Penalties [cycles]

	Predicted	
Outcome	**Taken**	**Not Taken**
Taken	1	2
Not Taken	2	0

3.3.3.5 ILP Model of Jump Penalties

The WCET of a basic block b_i does not only depend on its own WCET and the WCET of the outgoing paths starting at b_i, but also on possible jump penalties resulting from rearranging the order of blocks inside a function.

By default, the WCET for a single execution of a basic block b_i also comprises possible jump penalties of unconditional and conditional jump instructions. Table 3.1 depicts the resulting penalty cycles for which the pipeline is stalled after processing a branch instruction which was predicted taken or not and the real outcome during execution. As can be seen, the worst case is a mispredicted branch causing two cycles pipeline stall.

The execution time of a basic block determined by a static timing analyzer always includes penalty cycles caused by the branch prediction according to Table 3.1. In order to simplify the jump penalty constraints presented in the following, the pure execution time of each basic block without such penalties is required. Thus, the penalty cycles of jump instructions depending on the prediction an the real outcome are distilled in advance. Then, the costs C_i of a basic block b_i used in Equations (3.1) – (3.3) thus have to be redefined:

- If block b_i does not end with a jump instruction, its costs C_i equal the WCET for a single execution.
- For a block b_i with an unconditional jump as last instruction, two cycles are subtracted from the WCET to derive the costs C_i. These two cycles are composed of one cycle for executing the jump and of one cycle pipeline stall for determining the jump target.
- The WCET of a block b_i ending with a conditional jump instruction is determined by evaluating all outgoing edges to find the edge resulting in the highest execution time of b_i. Depending on whether this is the fall- or pass-through edge, it is also known if the branch is taken or not in the worst case. To determine the jump penalties for a conditional branch instruction from Table 3.1, the bit width and the jump direction has to be evaluated, too. If, for instance, a conditional branch was mispredicted, two cycles pipeline stall occur. In contrast to an unconditional jump, only these two cycles are subtracted from the WCET of the corresponding block b_i in order to calculate the costs C_i. This is done since the conditional jump itself cannot be removed.

Example 3.3 (*Unconditional jump instruction*)

Two basic blocks b_i and b_j are connected by an unconditional jump instruction bypassing a number of basic blocks as depicted in Figure 3.5b, p. 48.

If b_i has an execution time for a single execution of $c_i = 20$ cycles, two cycles are subtracted in order to distill the influence of the jump instruction at the end of b_i. Thus, the block's costs are redefined to $c_i = 18$ cycles.

Example 3.4 (*Conditional jump instruction*)

A basic block b_i ends with a conditional jump instruction which branches to a block b_k if a test condition applies. Otherwise, b_i's implicit successor b_j as depicted in Figure 3.5d, p. 48, is executed.

It is assumed that the branch is predicted taken and the jump target b_k is executed leading to one penalty cycle where the pipeline is stalled. If b_i has an execution time for a single execution of $c_i = 20$ cycles, the penalty cycle is subtracted in order to distill the influence of the jump instruction at the end of b_i. Thus, the block's costs are redefined to $c_1 = 19$ cycles.

To determine the jump penalties as ILP constraints, the different jump scenarios which are depicted in Figure 3.5 and their impact on the branch prediction of the employed processor have to be modeled: the simple case in Figure 3.5a is an implicit edge between two contiguous basic blocks b_i and b_j where b_i does not end with a jump instruction. If the ILP computes to not allocate these blocks contiguously ($x_i \neq x_j - 1$), then an unconditional jump has to be inserted at the end of b_i resulting in two cycles penalty:

$$jp^i_{impl} = 2 - 2 * (b_i \circ b_j) \tag{3.7}$$

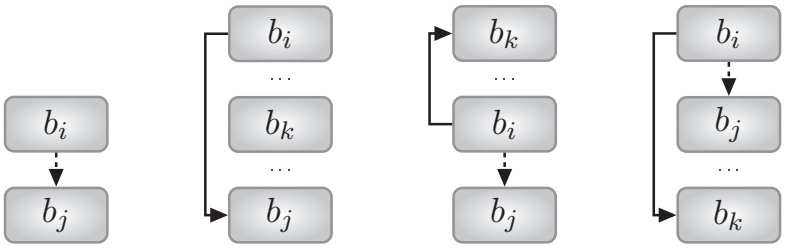
In Equation (3.7), the operator $\circ$ checks if two blocks are contiguous in memory by evaluating the corresponding decision variables:

$$(b_i \circ b_j) = \begin{cases} 1 & \text{if } x_i = x_j - 1 \\ 0 & \text{else} \end{cases} \tag{3.8}$$

The way how the $\circ$ operator is modeled within the ILP is omitted at this point. Refer to Equation (A.3) in the Appendix for a detailed specification of its constraints.

An unconditional branch, as shown in Figure 3.5b, also connects exactly two basic blocks b_i and b_j and usually bypasses a number of other basic blocks with a jump instruction. Due to the fact that the jump costs were distilled from the costs C_i in advance, this case can be also handled by Equation (3.7): If the ILP decides to allocate these blocks contiguously, nothing has to be done since removing the jump results in jump scenario 3.5a. Otherwise, two cycles penalty have to be readded for each execution of b_i.

Compared to the unconditional branch instructions, conditional jumps require a rather complex modeling of jump constraints: The static branch prediction of the



(a) Implicit (b) Unconditional (c) Conditional Backward (d) Conditional Forward

Figure 3.5: Typical Jump Scenarios

TriCore TC1796 distinguishes between 16 bit conditional jumps which are always predicted taken and 32 bit conditional jumps where the prediction depends on the jump direction. A 32 bit conditional jump with backward displacement (cf. Figure 3.5c) is predicted taken whereas the same instruction with forward displacement (cf. Figure 3.5d) is predicted not taken.

Jumps with 16 bit width are always predicted taken; in compliance with the second column of Table 3.1 either one cycle for a correctly predicted (pass-through edge) or two cycles for a mispredicted branch (fall-through edge) have to be added as penalty. Since the costs C_i of a block b_i are free of any jump penalties, the jump penalty constraints for both successors can now be handled in the same way. For each edge to successor $b_{succ} \in \{b_j, b_k\}$, a separate constraint determines the jump penalty depending on whether blocks b_i, b_{succ} are contiguous in memory or not:

$$jp_{cond16}^{i,succ} = 1 + (b_i \circ b_{succ}) \tag{3.9}$$

If $b_i \rightarrow b_{succ}$ is not contiguous (pass-through edge), the penalty for visiting b_{succ} from b_i is only one cycle for a correctly predicted branch. But if the edge to b_{succ} is the fall-through edge ($(b_i \circ b_{succ}) = 1$), a second cycle for a mispredicted branch is added to the penalty.

In contrast, 32 bit jumps require more complex constraints since the prediction of the target depends on the direction of the jump. For a backward jump which can be seen in Figure 3.5c, the second column of Table 3.1 has to be modeled whereas for a forward jump, the third column applies. As done for the 16 bit jump penalties, there is no need to care about the initial order of the basic blocks and the resulting jump penalties due to the distilled costs C_i. Instead, the four possible jump scenarios for a block b_i and its successors b_j and b_k are modeled as constraints. In the following, the jump penalties of edge $b_i \rightarrow b_j$ are presented, but edge $b_i \rightarrow b_k$ is modeled analogously:

1. Blocks b_i and b_j are contiguous (fall-through edge) and jumping to b_k results in forward displacement (predicted not taken). According to Table 3.1, the correctly predicted implicit edge results in no penalties:

$$jp^{i,j}_{cond32} \geq 0 * (b_i \circ b_j \wedge x_i < x_k) \tag{3.10}$$

Of course, this constraint is not added to the ILP but depicted for the sake of completeness. At this point, the interested reader is referred to Equations (A.4) and (A.2), Appendix p. 146, which describe the modeling of the *and* as well as the *less-than* operator.

2. Blocks b_i and b_j are contiguous (fall-through edge) and jumping to b_k results in backward displacement (predicted taken). If the fall-through edge is visited although the jump was predicted taken, two penalty cycles are the result:

$$jp^{i,j}_{cond32} \geq 2 * (b_i \circ b_j \wedge x_i > x_k) \tag{3.11}$$

3. Blocks b_i and b_k are contiguous and jumping to b_j (pass-through edge) results in forward displacement (predicted not taken). Since the fall-through target b_k is predicted to be executed, jumping to b_j would result in two cycles penalty:

$$jp^{i,j}_{cond32} \geq 2 * (b_i \circ b_k \wedge x_i < x_j) \tag{3.12}$$

4. Blocks b_i and b_k are contiguous and jumping to b_j (pass-through edge) results in backward displacement (predicted taken). If jumping to b_j is predicted correctly, only one cycle penalty has to be added:

$$jp^{i,j}_{cond32} \geq 1 * (b_i \circ b_k \wedge x_i > x_j) \tag{3.13}$$

Depending on the jump scenario (JS) of a basic block b_i, the overall jump penalty jp_i is defined as follows:

$$jp_i = \begin{cases} jp^i_{impl} & \text{if JS of } b_i \text{ is } \textit{implicit} \text{ or initially } \textit{unconditional} \\ jp^{i,succ}_{cond16} & \text{if JS of } b_i \text{ is } \textit{conditional} \text{ 16 bit} \\ jp^{i,succ}_{cond32} & \text{if JS of } b_i \text{ is } \textit{conditional} \text{ 32 bit} \\ 0 & \text{else} \end{cases} \tag{3.14}$$

The jump penalties are used to extend the basic control flow constraints defined in Equations (3.1) and (3.2):

$$w^L_{exit} = C^L_{exit} + jp^L_{exit} \tag{3.15}$$

$$\begin{aligned} \forall b_i \in V \setminus \{b^L_{exit}\} : \ \forall (b_i, b_{succ}) \in E : \\ w_i \geq w_{succ} + C_i + jp_i \end{aligned} \tag{3.16}$$

3.3.3.6 ILP Model of the Global Control Flow

Up to this point, Equations (3.4) - (3.16) only model the intra-procedural control flow of a single function F within the ILP. Without loss of generality, one dedicated entry block b^F_{entry} is assumed as first block of F. For b^F_{entry}, the ILP variable w^F_{entry} denotes the WCET of any path starting at b^F_{entry} for a single execution of F.

However, some basic block b_i of a function F' may contain a call to function F. In this situation, F's WCET represented by variable w^F_{entry} has to be added to the WCET of block b_i. Thus, the control flow constraint in Equation (3.16) is extended by w^F_{entry}, representing F's WCET, if block b_i calls F:

$$\begin{aligned} &\forall b_i \in V \setminus \{b^L_{exit}\} : \ \forall (b_i, b_{succ}) \in E : \\ &\quad w_i \geq w_{succ} + C_i + jp^i_{impl} + w^F_{entry} \end{aligned} \tag{3.17}$$

Although a call is not counted as jump instruction, the modeling of jump penalties is required for the control flow edge to the implicit successor. If the succeeding basic block is moved away, an implicit jump has to be inserted to restore the control flow.

Example 3.5 (*Global Control Flow*)

Assumed that basic block b_j of Example 3.1 calls a function `foo`*, the WCET of* `foo`*'s entry basic block $b^{\texttt{foo}}_{entry}$ is added to the costs of b_j according to Equation* (3.17)*. Only the constraint of Example 3.1 modeling the WCET of b_j thus has to be extended:*

$$w_j \ = \ C_j + w_l + w^{\texttt{foo}}_{entry} \ = \ 30 + 15 + w^{\texttt{foo}}_{entry} \ = \ 45 + w^{\texttt{foo}}_{entry}$$

The basis for Equation (3.17) stems from [62].

3.3.3.7 Objective Function

The overall goal of the ILP is to minimize a program's WCET by rearranging the order of basic blocks inside a function. Due to the nature of Equations (3.16) and (3.17), variable w^F_{entry} corresponds to the WCET of function F including the WCETs of all functions called by F extended by possible jump penalties. Function `main` is the unique entry point of an entire program; hence, variable $w^{\texttt{main}}_{entry}$ denotes the overall WCET of the program. As a consequence, the value of this variable has to be minimized by the ILP:

$$w^{\texttt{main}}_{entry} \rightsquigarrow min. \tag{3.18}$$

3.3.4 Evaluation

This section evaluates the performance of the WCET-driven branch prediction aware code positioning algorithms applied to real-life benchmarks. In Section 3.3.4.1, the experimental environment which is employed to perform evaluations is presented. Section 3.3.4.2 discusses the WCET reductions achieved by the code positioning algorithms described in Section 3.3.3, whereas Section 3.3.4.3 discusses the achieved ACET reductions. Finally, Section 3.3.4.4 deals with the computational complexity of the proposed optimizations.

3.3.4.1 Experimental Environment

For benchmarking, the optimization level *O3* is used for which the WCC compiler (cf. Figure 2.2) applies all 33 optimizations listed in Table 2.1, p. 27, in order to evaluate the performance of the new algorithms on highly optimized code. The compiler generates code for the Infineon TriCore TC1796 processor. This TriCore v1.3 family is equipped with a static branch prediction unit for which equations (3.7) - (3.16) are tailored.

For all measurements, 15 benchmarks stemming from the benchmark suites *Mediabench* [65], *MiBench* [47], *MRTC* [46] and *UTDSP* [115] were used. The number of benchmarks was limited since the evolutionary optimization algorithm would otherwise require several weeks of optimization runtime as discussed in Section 3.3.4.4. As listed in Table 3.2, the number of functions of the benchmarks ranges 1 (*g721_decoder*) up to 28 (*g723_encoder*) whereas the overall number of basic blocks ranges from 10 up to 380 (*g723_encoder*).

Today's embedded systems are equipped with main memories in megabyte ranges. Nevertheless, all evaluations are performed with the program code residing in the fast SPM in order to avoid undesired side-effects by hardly predictable access latencies of FLASH memories due to their page buffers. This enables comparable results for both optimization algorithms. Otherwise, the evolutionary algorithm would have a slight advantage since the repetitive creation and evaluation of individuals implicitly considers memory hierarchy effects by repetitive WCET analyses. In contrast, these effects are very difficult to express adequately within an ILP-based optimization.

In order to evaluate the achievable WCET reduction, the evolutionary approach is invoked with an initial population of $\alpha = 20$ individuals. Each offspring generation has $\mu = 20$ parents and also comprises $\lambda = 20$ individuals. The maximum number of generations amounts to $maxGen = 20$.

In order to derive the basic block costs C_i required for the ILP model described in Sections 3.3.3.3 to 3.3.3.7, a single WCET analysis employing *aiT* is performed. The loop iteration counts, however, are directly read from the *flow facts* (cf. Section 2.4.5) attached to the basic blocks representing the loop headers. To solve the generated ILP, *IBM ILOG CPLEX* [56] is employed which is a sophisticated solver for *integer-linear programming* problems.

Table 3.2: Benchmark Characteristics

Benchmark	*#Functions*	*#Basic Blocks*	*Code Size* [bytes]
basicmath	20	282	10,088
binarysearch	2	10	1,712
countnegative	3	19	1,928
crc	3	28	2,136
fir_256_64	2	10	1,704
g721_decoder	1	14	1,720
g723_encoder	28	380	6,912
janne_complex	2	15	1,680
lmsfir_32_64	2	18	1,960
ndes	6	87	3,712
prime	3	14	1,744
qurt	3	26	2,208
select	2	31	2,488
selection_sort	2	16	1,712
sqrt	2	14	1,800
Average	5	64	2,900

3.3.4.2 WCET Estimations

Figure 3.6 depicts the results achieved by the branch prediction aware code positioning algorithms for the considered 15 benchmarks. For each benchmark, the left bar represents the result for the evolutionary code positioning technique, whereas the right bar represents the result if the ILP-based algorithm is applied. The 100% line is equal to the estimated WCET of the benchmarks compiled with the optimization level *O3* without code positioning (the code layout mainly matches the logical order found in the *C* source code). The bars depict the WCET_{est} of the optimized program computed by the static WCET analyzer as percentage of its "unoptimized" version.

The evolutionary algorithm reduces the WCET_{est} of the benchmarks by up to 24.7% (*prime* benchmark). For the same benchmark, the ILP-based algorithm is able to achieve WCET_{est} reductions of up to 20.0%. Significant WCET_{est} reductions could be achieved for almost all benchmarks except for *basicmath*. Here, the EA achieved only a marginal WCET_{est} reduction of 0.6% whereas the ILP-based approach did not achieve any improvement. On average, the WCET_{est} for all benchmarks was reduced by 8.9% by applying the evolutionary approach and by 6.7% for the ILP-based optimization, respectively.

Table 3.3 shows the ratio of executed unconditional (columns labeled "Uncond.") and mispredicted conditional jump instructions (columns labeled "Mispr.") on the

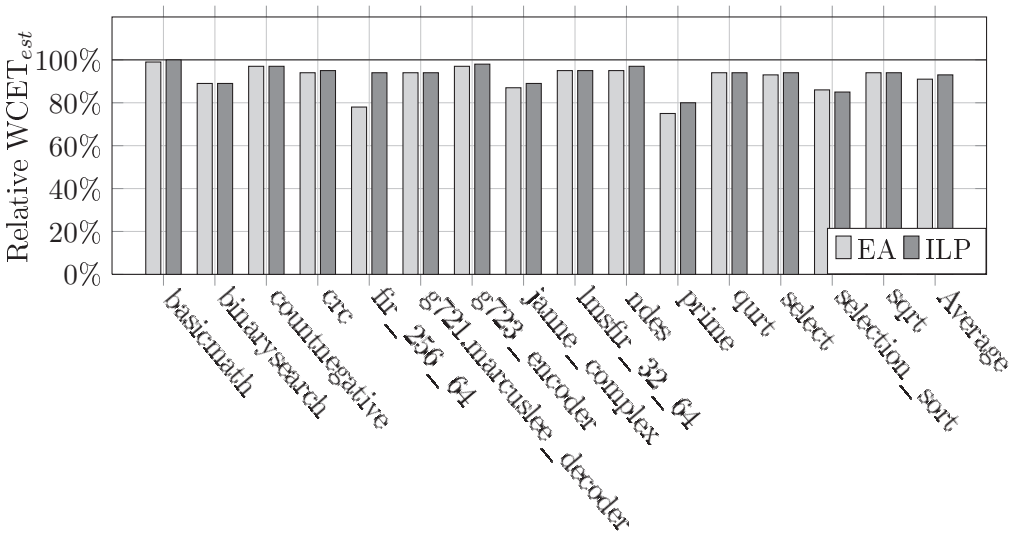


Figure 3.6: Relative WCET$_{est}$s after Code Positioning Optimizations

WCEP. For each benchmark, values for the unoptimized version, the evolutionary and the ILP-based approach were collected. Here, 100% equals the overall number of executed jump instructions on the WCEP (conditional and unconditional) in columns labeled with "#JI". Multiple executions of a jump are counted multiple times.

Considering the amount of mispredicted branches, both algorithms perform best for the *selection_sort* benchmark where reductions of 56% (EA) and 55.1% (ILP) were achieved. Conversely, unconditional jumps had to be inserted in order to correct the control flow. They amount to 21.3% (EA) and 20% (ILP) of the overall executed number of jumps. Thereby, WCET$_{est}$ reductions of 14.4% and 14.6% were achieved, respectively.

The amount of unconditional jumps could be decreased by 32.4% and 33% for the benchmark *fir_256_64* by applying the evolutionary and the ILP-based approach, respectively. Even though considerable reductions can be achieved, in the majority of cases, the number of executed unconditional jumps was increased by up to 33.3% (*g721.marcuslee_decoder, EA)* and 33.9% (*ndes*, ILP).

On average, the number of mispredicted branches was reduced by 16.8% (EA) and 21.4% (ILP) whereas the unconditional jumps were increased by 4.1% (EA) and 6.7% (ILP).

Corner cases: For the benchmark with the highest WCET$_{est}$ reductions (*prime*), it turned out that both reordering algorithms were able to eliminate almost all mispredicted branches. No unconditional jumps are executed at all. Nevertheless, the EA algorithm was able to outperform the ILP-based approach by 4.7% w. r. t. WCET$_{est}$ reduction.

Table 3.3: Ratio of unconditional and mispredicted jumps

	Unoptimized			EA			ILP		
	#JI	*Ucond.*	*Mispr.*	*#JI*	*Ucond.*	*Mispr.*	*#JI*	*Ucond.*	*Mispr.*
basicmath	76691	0.0%	0.5%	77407	0.9%	0.6%	76691	0.0%	0.5%
binarysearch	15	0.0%	66.7%	14	14.3%	35.7%	17	29.4%	29.4%
countnegative	1240	32.3%	0.1%	1041	19.3%	0.1%	840	0.0%	0.1%
crc	8878	23.1%	24.0%	8892	23.2%	24.0%	9388	27.3%	16.4%
fir_256_64	768	33.2%	0.1%	517	0.8%	49.3%	514	0.2%	0.2%
g721.marcuslee	14442	0.0%	50.0%	21664	33.3%	0.0%	16849	14.3%	14.3%
g723_encoder	95980	2.3%	23.5%	96133	2.5%	23.8%	141217	35.0%	9.2%
janne_complex	70	5.7%	47.1%	75	12.0%	22.7%	71	7.0%	16.9%
lmsfir_32_64	14802	17.0%	24.5%	14962	17.9%	17.0%	14931	17.7%	1.1%
ndes	4369	0.1%	74.3%	6007	27.3%	41.6%	6617	34.0%	33.3%
prime	865	0.0%	49.7%	865	0.0%	0.2%	866	0.0%	0.1%
qurt	240	0.0%	27.5%	246	2.4%	1.2%	277	13.4%	13.4%
select	969	0.0%	1.8%	970	0.1%	1.8%	1004	3.5%	1.8%
selection_sort	341160	0.0%	74.9%	433315	21.3%	18.9%	425300	20.0%	19.8%
sqrt	547	0.0%	24.3%	547	0.0%	0.0%	620	11.8%	11.8%
Average	37402	7.6%	32.6%	44177	11.7%	15.8%	46347	14.2%	11.2%

This behavior is caused by the impact of the resulting memory layout of the modified program on the WCET_{est}: it is, for instance, worthwhile to align loop headers at the beginning of memory lines. Otherwise, the crossing of memory lines is more frequent. This often results in a decreased performance since multiple memory lines have to be fetched in order to execute the loop header. The ILP is not aware of any memory addresses during the rearranging of basic blocks with the result that line crossing effects cannot be modeled. In contrast, the evolutionary approach always implicitly takes the impact of memory layout modifications on the WCET_{est} into account by evaluating the fitness of each newly created individual using *aiT*.

For the *fir_256_64* benchmark, both algorithms were able to remove almost all executed unconditional jump instructions (0.8% respectively 0.2% remaining). But the evolutionary algorithms was able to outperform the ILP optimization by 16% WCET_{est} reduction (77.7% vs. 93.7% resulting WCET_{est}) although the number of mispredicted branches is increased by 33%. The ILP generates a basic block order where 99.6% of the conditional branches are correctly predicted taken, each resulting in one cycle pipeline stall (cf. Table 3.1). The EA, however, determines a memory layout where half of the branches are correctly predicted not taken resulting in no penalty cycles. The remaining conditional jumps were mispredicted, each resulting in two cycles penalty. Since the executed jumps induce the same amount of overall pipeline stall cycles, the evolutionary algorithm only performs better due to a memory layout causing less line crossings of instructions.

Although the evolutionary approach should have a small advantage compared to the ILP-based optimization, there are cases, for instance the benchmark *selection_sort*, where the ILP-based approach performs better (in finite time): the ILP optimization explicitly models absolute positions of basic blocks and the influence of

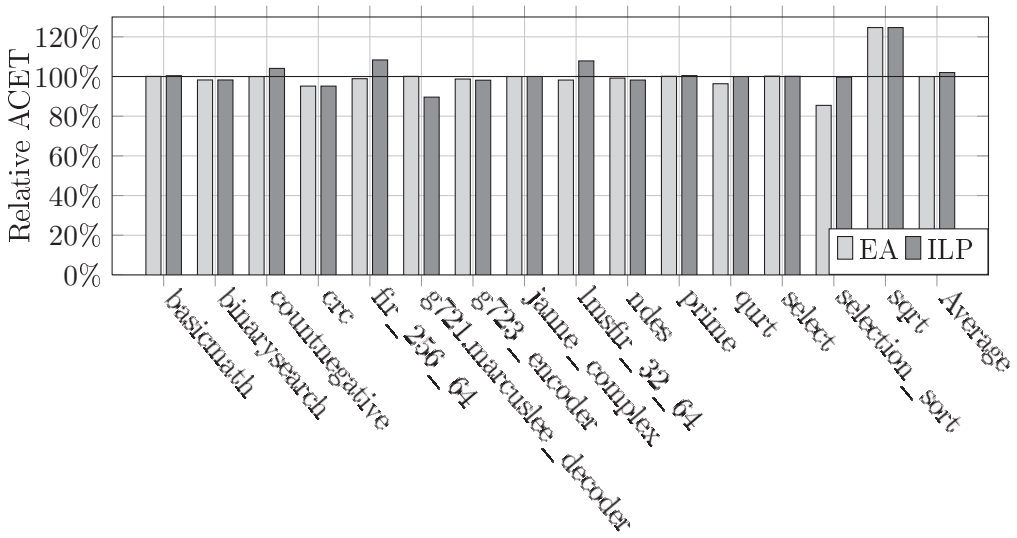


Figure 3.7: Relative ACETs after Code Positioning Optimizations

possible branch penalties on the WCET. The corresponding constraints are always considered during solving the equations. Thus, the complexity only depends on the number of constraints which in turn depend on the control flow of a program. Hence, the overhead for solving the ILP is depending on the total number of basic blocks but not on the number of blocks which have to be moved during optimization.

In contrast to the ILP model, the evolutionary approach can only apply small modifications to the order of basic blocks of a function by employing the operators crossover and mutation when a new offspring individual is created. Thus, for benchmarks with complex control flows with a lot of control flow edges, a large number of offspring individuals has to be created until the space of valid solutions can be explored extensively in order to find an improved solution. The *selection_sort* benchmark is an example where the optimal result is typically found after 40 generations, leading to up to 800 WCET analyses.

Such an insufficiency could be tackled by tuning the evolution parameters; for instance, each position of the solution vector could be mutated with a certain probability instead of mutating only a single position. In this thesis, the evolutionary approach only serves as a case study if reordering basic blocks w. r. t. the WCET of a program pays off. Another aim is to obtain WCET reference data for a comparison with the ILP-based approach. Thus, improving the evolutionary approach is omitted at this point.

Both algorithms are not able to achieve appreciable WCET reductions for the *basicmath* benchmark. The initial order of basic blocks generated by WCC's code selector already supports the branch prediction such that only 0.5% of the branches were mispredicted. Since there are less than 0.1% unconditional jump instructions on the WCEP, there is almost no optimization potential w. r. t. the order of basic blocks.

3.3.4.3 ACET Estimations

Figure 3.7 depicts the impact of the branch prediction aware code positioning algorithms on the ACET. The commercial, cycle-true instruction set simulator CoMET [109] was employed to measure ACETs. Once again, the bars depict the resulting ACET of the optimized program as percentage of its "unoptimized" version. For each benchmark, the left bar represents the result achieved by the evolutionary code positioning technique, whereas the right bar represents the result if the ILP-based algorithm is applied. The 100% line is equal to the ACET of the benchmarks compiled with the optimization level *O3* without any code positioning optimization (the code layout matches the logical order found in the *C* source code).

The evolutionary algorithm was able to decrease the ACET of the considered benchmarks by up to 14.4% (*selection_sort*). But there are also cases, where the ACET is increased (*sqrt*, 24.7% increase). On average, a marginal ACET decrease of 0.3% can be achieved.

Evaluating the ILP-based optimization shows that the algorithm does not perform better than the EA. Here, too, ACET reductions can be observed (only up to 10.4% for *g721.marcuslee_decoder*). But once again, there are cases of increased ACETs (up to 24.7% for *sqrt*). On average, the ACET is even worsened by 1.7%.

Comparing Figures 3.6 and 3.7 shows that both algorithms behave completely different for the estimated WCET and the ACET: the WCET-driven branch prediction aware code positioning is best suited to achieve WCET reductions but performs worse for ACET optimization. This is caused by the fact that the WCET serves as metric during the optimization of the programs. The EA evaluates the fitness of individuals by invoking a static timing analyzer. The ILP-based optimization employs WCETs of basic blocks as well as worst-case execution frequencies to set up the constraints modeling the objective function. Hence, the impact of any reordering on the ACET is not taken into account.

The WCET_{est} of the benchmark *sqrt*, for instance, can be decreased by 6% whereas the ACET is worsened. In *sqrt*, a loop performs square root computations on floats. For average-case scenarios, this loop exits usually after few iterations. Regardless of this fact, a WCET analyzer has to assume the maximum iteration count of the loop as worst case. Thus, the WCEP is different from the most frequently used path. The positions of the benchmark's basic blocks are changed in order to shorten the WCEP. In order to correct the control flow, unconditional jumps are inserted on that path that is most frequently used in an average-case scenario, but which (in this case) does not contribute to the WCEP. The inserted jumps lead to an ACET increase of 24.7%.

These observations obtained comparing the ACET and WCET performance of WCET-tailored optimizations conform to the observations presented in [31].

3.3.4.4 Optimization Time

To consider the optimization time, an Intel Xeon E5506 (2.13 GHz) was utilized. Most of the time necessary for the novel WCET-driven branch prediction aware code positioning algorithms was consumed by the WCET analyses using *aiT*. The maximal number of WCET analyses during an evolutionary optimization run is equal to the number of created individuals and amounts to

$$\#Analyses_{WCET} \leq \alpha + \lambda * (maxGen - 1) = 400 \tag{3.19}$$

where α is the size of the initial population and λ the number of offspring individuals. For the ILP-based approach, only a single WCET analysis is necessary to determine the costs C_i for each basic block b_i (cf. Section 3.3.3.2).

For a single WCET analysis, up to 20 CPU minutes are required for the *basicmath* benchmark. Thereby, the evolutionary approach requires almost 2 days for the optimization of this benchmark on a CPU single core which is not feasible in practice. The ILP-based optimization, in contrast, merely spends 20 minutes for the one required WCET analysis which is highly suitable for most application scenarios. The immense WCET analysis time for the evolutionary approach is the reason why the number of benchmarks had to be limited to 15 in order to avoid optimization times of months.

The complexity of solving the ILPs generated by the optimization discussed in Section 3.3.3.2 is of no practical relevance. For a CFG with n nodes, the ILP has a size of $O(n^2)$ constraints and variables. The employed ILP solver *CPLEX* takes up to 2 CPU minutes (*basicmath*) but mostly terminates within a few seconds for the considered benchmarks. Compared to the WCET analysis required to determine the cost constants C_i for each basic block and the immense time required for the evolutionary algorithm, these values are convenient.

The publication [93] has arisen from the techniques presented in this section.

3.4 Cache-aware Memory Content Selection

Memory content selection techniques aim at improving a certain objective by intelligently allocating memory objects to available memories. In order to reduce the average-case runtime of a program, frequently executed code blocks or used data objects are typically allocated to faster memories. The WCET-driven scratchpad allocation presented in [32] is an example for memory content selection. This chapter, however, addresses the allocation of memory content with regard to the effect on the I-cache performance and the resulting WCET_{est} of a program to optimize.

In systems equipped with caches, the latency of an access to a certain main memory address highly depends on the contents of the cache. If an instruction to be fetched already resides in the cache, a *cache hit* occurs and the fetch can be usually performed within one processor clock cycle. Otherwise, the access results in

a *cache miss*. The required data then has to be fetched from the slow main memory (e.g. Flash) leading to penalty cycles depending on the processor and memory architecture.

The disadvantage of systems with caches is the limited predictability. It is hard to determine if an access results in a cache hit or miss and thus, it is hard to predict the execution time of a program executed from a cached memory. This is caused by the fact that caches only speed up the execution if a program tends to reuse instructions in the near future. If, however, the code of a program is not suitably arranged in the address space and the memory accesses are random or widely spread over the address space, the performance can be also decreased by the usage of a cache. Hence, static analysis techniques have been developed to allow safe predictions of a cache's impact on the worst-case performance of a system [113] in order to estimate tight bounds for the WCET of a program.

Intelligent allocation of beneficial functions to cached memory areas and unfavorable functions to non-cached memory areas can ensure that functions whose WCET highly profits from a cache are not evicted from the cache by functions with a low benefit. This can lead to a faster execution and a decreased WCET due to a dramatically decreased number of *real* cache misses. Furthermore, it is possible to reduce the overestimation of the WCET which is introduced by pessimistic assumptions concerning the I-cache performance: If the memory access pattern cannot be determined at a certain point of the program execution, a static WCET analyzer has to consider an *assumed* cache miss and invalidated cache content. By allocating only a promising subset of functions to cached memory areas, the amount of unpredictable cache accesses can be decreased. As a result, the cache analysis assumes less cache misses caused by such unpredictable accesses and a tighter bound for the WCET can be computed.

In the following, a novel WCET-driven cache-aware memory content selection algorithm is presented which selects functions to be placed in a cached memory area in order to improve the worst-case I-cache performance. The proposed algorithm uses a greedy approach and evaluates the impact of executing a function from a cached memory area on the WCET. The algorithm always selects the function with the largest profit and keeps track of changing WCEPs by subsequently performing a WCET analysis.

This chapter is organized as follows: In the next section, cache-aware memory allocation techniques are motivated by an example, followed by an overview of related work in Section 3.4.2. Section 3.4.3 presents our new WCET-driven cache-aware memory content selection algorithm. An evaluation of the performance which is achieved by our WCET-aware memory allocation algorithm is presented in Section 3.4.4.

```
1 void foo1() {
2  for(i=0; i<10; i++) {
3   foo2();
4   foo3();
5  }
6  ...
7 }
```

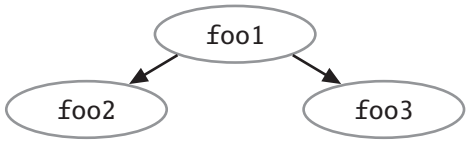


Figure 3.8: Exemplary Program and resulting Call Graph

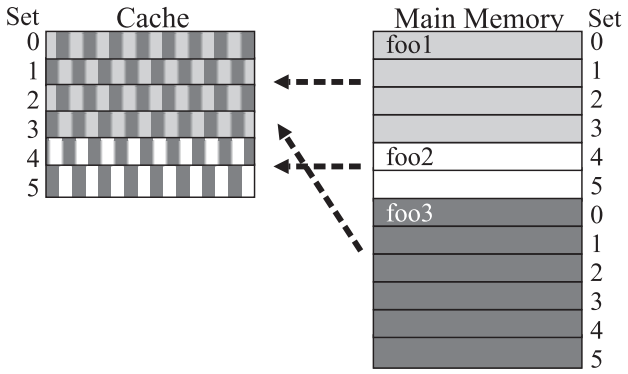


Figure 3.9: Cache thrashed by mutual Evictions of Functions

3.4.1 Motivating Example

Nowadays, embedded systems are equipped with caches in kilobyte ranges, typically from 1 kB up to 16 or 32 kB. Compared to growing main memories in megabyte ranges, caches are rather small. The I-cache controller tries to keep copies of frequently executed memory lines containing sequences of instructions as cache content for a faster access. If a cache miss occurs, a complete cache line containing the addressed item is fetched from the main memory and possibly evicts valid content from the cache. The amount of cache misses highly depends on the ratio of cache to memory size, the cache replacement policy and the structure of the executed program. A high amount of cache misses implies costly reloading of content from the slow main memory and leads to a high number of penalty cycles due to pipeline stalls.

Figure 3.8 depicts an examplary program for which a high number of cache misses can occur: The left-hand side shows the C code of a function `foo1` containing a loop which calls functions `foo2` and `foo3`. The resulting call graph of this simple program is shown on the right side. If the functions are consecutively arranged in the memory, `foo1` and `foo2` can be simultaneously stored in the cache (cf. Figure 3.9). Since the cache capacity is not large enough to store the whole program, `foo3` evicts the complete cache content during its execution. Thus, a lot of conflict misses

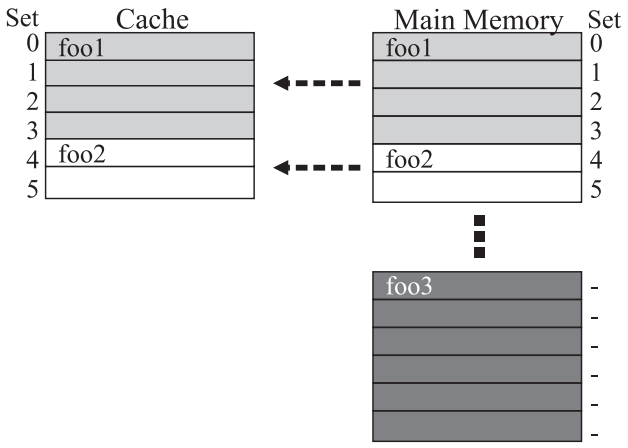


Figure 3.10: Exploiting non-cached Memory Areas to avoid Cache Thrashing

occur and the complete cache has to be refilled twice during each iteration of the loop in `foo1`. This might lead to a dramatically increased execution time caused by pipeline stalls due to a high number of cache misses.

Many embedded systems have parameterizable caches and memory layouts which allow that parts of the address space can be included or excluded from caching. An optimizing compiler can exploit such a memory system and allocate frequently called functions to cached memory regions and rarely used functions to non-cached memory regions. This strategy can ensure that functions which highly benefit from a cached execution cannot be evicted from the cache by functions with a lower benefit.

In the following we assume the WCETs for functions `foo1`, `foo2` and `foo3` as depicted in Table 3.4. WCET_{Flash} is the overall WCET of a function `f` if the entire function is executed from a non-cached memory area summed up over all iterations of the loop in `foo1`. WCET_{Cached} is `f`'s WCET if it is executed from a cached memory area. Obviously, it is more favorable if functions `foo1` and `foo2` can be kept in the cache since the accumulated WCET of `foo1` and `foo2` is decreased from 1040 cycles to 665 cycles. Overall, the WCET is reduced by 375 cycles compared to 131 if `foo3` is kept in the cache. An optimized memory layout for the example in Figure 3.8 is shown in Figure 3.10. Function `foo3` is moved to a non-cached memory area in order to prevent a constantly repetitive eviction of function `foo1` and `foo2`.

Table 3.4: WCETs for Functions depending on the Memory Region

Function	WCET_{Flash}	WCET_{Cached}
`foo1`	350	195
`foo2`	690	470
`foo3`	500	369

3.4.2 Related Work

Theiling et al. present static WCET analyses for systems with caches based on their research in [113]. They separate microarchitectural analysis from path analysis in order to manage the overall analysis complexity. This way, a fast and still precise WCET analysis is feasible.

Work presented in [106] by Suhendra et al. examines a combination of locking and partitioning of shared caches on multi-core architectures in order to guarantee a predictable system behavior. The proposed algorithms are not WCET-aware since their decisions are not based on any WCET values; nevertheless, the authors evaluate the impact of their caching schemes on the worst-case application performance.

In [36], Falk et al. present a cache locking algorithm which explicitly takes the worst-case execution path into account during each step of the optimization procedure. This way, they can make sure that always those parts of the code are locked in the I-cache that lead to the highest WCET reduction. Puaut et al. [99] propose locked instruction caches in multi-task real-time systems. They propose two low-complexity algorithms for cache content selection. A drawback of statically locking the cache content is that the dynamic behavior of the cache is lost. In contrast to the techniques proposed in this chapter, code is no more automatically loaded into the cache. Thus, code which is not locked into the cache cannot profit from it anymore.

Vera et al. [117] combine cache partitioning, dynamic cache locking and static cache analysis of data caches to achieve predictability in preemptive systems. This eliminates overestimation and allows to approximate the worst-case memory performance, however, unlike our new approach, they are not able to explicitly optimize the WCET of a system.

In [42], a technique rearranging the positions of tasks to improve the cache performance is presented. Gebhard et al. propose to evaluate the interdependency relation of tasks in order to determine a memory layout which maximizes the number of persistent cache sets for each task. In contrast to the optimization presented in this chapter, intra-task cache conflicts cannot be optimized since the order or allocation of functions inside a task stays untouched.

A technique for procedure placement to reduce the cache miss ratio of programs is presented by Guillon et al. in [45]. The authors provide an optimal algorithm for memory placement which is improved regarding the unavoidable code size increase caused by gaps in the address space. In contrast to our optimizations, the presented approach does not take the WCET as metric into account.

Another technique for procedure positioning is presented by Lokuciejewski et al. in [77]. The authors propose an iterative algorithm which evaluates the worst-case calling frequencies and takes the WCET as metric into account. Unlike the work presented in this chapter, their algorithm exploits preloading if parts of contiguous functions occupy the same cache line but suffers from small caches leading to increased cache miss rates.

Falk et al. counteract the possible predictability problems of caches with a static allocation of program code to so-called scratchpad memories (*SPM*) [32]. They employ *integer-linear programming* to select the optimal content of the SPM w. r. t. the program's WCET. Static SPM allocation has the disadvantage that the content is fixed during the program's execution. Thus, only the code located in the SPM profits from an accelerated execution. Furthermore, jump instructions have to be inserted in order to keep the control flow consistent. Such a code modification introduces additional runtime overhead.

3.4.3 WCET-driven Memory Content Selection Algorithm

In the following, a novel algorithm for a WCET-driven cache-aware memory content selection is described which moves functions to different memory areas depending on their impact on the overall WCET.

As already stated in [77], even local code modifications can have a strong impact on the WCET of other parts of a program. This impact is hardly predictable without performing a complete static WCET-analysis of the program. Moreover, the situation becomes even more complicated if systems are equipped with caches. To deal with this handicap, a greedy optimization algorithm has been developed which moves single functions between memory areas and evaluates the influence on the WCET of the program to optimize by performing a WCET-analysis using *aiT*. The new assignment of functions to cached and non-cached memory areas acts as starting point for the next iteration which moves another function in case of a preceding WCET decrease. But if the WCET is increased, the last modification is rolled back in order to guarantee that the optimized program is never worse than the original code w. r. t. its WCET.

Algorithm 1 shows the pseudo code of the greedy algorithm requiring a set F of functions to be optimized as input. The first two lines define sets of functions which are decided to be placed in a cached memory region (set CF) or a non-cached memory region (set NCF). The variable S is initialized with the cache size (line 3) and acts as counter for the free cache in bytes.

The profit for a function f if it will be moved from a non-cached memory region to a cached memory region is calculated in line 4. The profit of a function f is defined as:

$$profit(f) = \frac{WCET(f) - cachedWCET(f)}{size(f)}$$

$WCET(f)$ is the WCET of function f if it is allocated to a non-cached memory region, while $cachedWCET(f)$ is f's WCET if it is executed from a cached memory.

Algorithm 1 Greedy WCET-driven Memory Content Selection Algorithm

Require: set<functions> F

set<functions> $CF = \emptyset$
set<functions> $NCF = \emptyset$
$S =$ cache size
calculateProfit(F)
while $(S > 0 \wedge F \neq \emptyset)$ **do**
  for $f \in F : max($ *getProfit*(f) $)$ **do**
    if (*getSize*$(f) > S$) **then**
      break;
    end if
    $CF = CF \cup f$
    $F = F \setminus f$
    $S = S -$ *getSize*(f)
  end for
end while
/* From now on evaluate the WCET trend since the cache is full */
WCET $lastWCET = evaluateWCET()$
for all $f \in F : max($ *getProfit*(f) $)$ **do**
  $CF = CF \cup f$
  $F = F \setminus f$
  WCET $newWCET = evaluateWCET()$
  if $(newWCET > lastWCET)$ **then**
    $CF = CF \setminus f$
    $NCF = NCF \cup f$
  else
    $lastWCET = newWCET$
    $calculateProfit($ F $)$
  end if
end for
return CF

To avoid side effects caused by cache conflict misses when two or more functions are mapped to the same cache lines, either the cache has to be large enough to store copies of all functions or only a subset of functions has to be placed into the cached memory region and analyzed at a time. For that, $cachedWCET(f)$ could be determined by moving each function separately into the cached memory and determining its WCET using a timing analyzer. However, this approach has the drawback that several static WCET analyses are required to compute $cachedWCET$ for all functions.

The static WCET analyzer *aiT* employed in this thesis is able to compute a program's WCET for systems with different cache sizes. To save time consumed by repetitive WCET analyses, a virtual large I-cache larger than the code size of the analyzed program is chosen for the evaluation. Moreover, we align each function at the start address of a cache line in order to achieve the same values as if all functions were analyzed separately.

In a few cases, a negative profit is detected. Then, function f incurs a WCET increase due to a cached execution and therefore is excluded from optimization.

The loop in lines 5-14 allocates functions to the cached memory as long as the overall code size still fits into the cache and not all functions are allocated. Inside the loop, the function with the highest profit (line 6) is removed from the set of unhandled functions and allocated to the cached memory region (lines 10-11). Line 12 keeps track of the allocated overall code size and lines 7-9 break the loop if the function with the highest profit does not fit into the cache anymore.

If the cache can store no more functions simultaneously, it is yet possible that allocating another function to a cached memory can decrease the WCET. Hence, lines 15-27 test if moving another function to the cached memory can achieve further reductions of the WCET.

Line 15 stores the current WCET of the program with the allocation decisions made in the preceding loop. The loop starting in line 16 iteratively moves one of the remaining functions with the highest profit to the cached memory region (line 17-18) and evaluates its effect on the WCET (line 19). If a WCET increase was detected in line 20, the following two lines roll the last change in the memory layout back and function f is stored in the set NCF of non-cached functions. Otherwise, the new decreased WCET serves as reference for the next iteration (line 24).

A new profit calculation is performed in line 25. Here, again, the WCET of the remaining functions $f \in F$ is taken into account if stored in non-cached memory as well as stored in cached memory. Therefore, we can recycle the WCET analysis results gathered in line 4 where $cachedWCET(f)$ was already determined for all functions in a cached memory. $WCET(f)$ can be reused from the results computed in line 19 where the remaining functions were stored in the non-cached memory.

$WCET(f)$ for a function f stored in a non-cached memory will be zero iff f is not lying on the WCEP. Thus, $profit(f) \leq 0$ is true and f is not considered as a possible candidate to move to a cached memory area during the next loop iteration. Hence, the algorithm inherently keeps track of a possible change of the WCEP.

Table 3.5: Benchmark Characteristics

Benchmark	*#Functions*	*#Basic Blocks*	*Code Size* [bytes]
adpcm	11	283	9920
g721_encode	28	380	6912
g723_encode	28	380	6291
gsm	38	467	21320
h264dec	4	142	14336
md5	12	67	5592
rijndael_decoder	6	94	12304
rijndael_encoder	8	109	12024
statemate	8	408	10688
v32.modem_benc.	9	140	6272
Average	13	210	10566

The algorithm terminates if all functions are either located in set CF for functions to be cached or set NCF of functions to exclude from caching.

Finally, all functions in set CF will be attached to a section which is allocated to a cached memory area by the linker. All remaining functions of set NCF belong to the `.text` section which is allocated to a non-cached area by default.

3.4.4 Evaluation

This section evaluates the capability of our WCET-driven cache-aware memory content selection algorithm applied to real-life benchmarks. For benchmarking, the optimization level *O3* was used for which the WCC compiler activates 33 different optimizations (cf. Table 2.1, p. 27) in order to evaluate the performance of our new algorithm on highly optimized code. The compiler emits code for the TC1796 processor with a 16 kB 2-way set associative I-cache and *laest recently used* (LRU) replacement policy. The TC1796 integrates a 2 MB program Flash as main memory which is mapped to two different addresses in the memory layout. The first memory region allows a cached access to the Flash whereas code executed from the other region in the address space is excluded from caching.

For our measurements, the 10 largest benchmarks from the benchmark suites *DSP Stone* [130], *Mediabench* [65], *MiBench* [47], *MRTC* [46], *NetBench* [85] and *UTDSP* [115] were utilized. As listed in Table 3.5, the number of functions of the benchmarks ranges from 4 (*h264dec*) up to 38 (*gsm*) whereas the code size ranges from 5.4 kB (*md5*) up to 20.8 kB (*gsm*).

Today's embedded systems are equipped with main memories in megabyte ranges and caches typically ranging from 1 kB up to 16 kB or at most 32 kB. Since the employed benchmarks would entirely fit into the cache, comparable results could

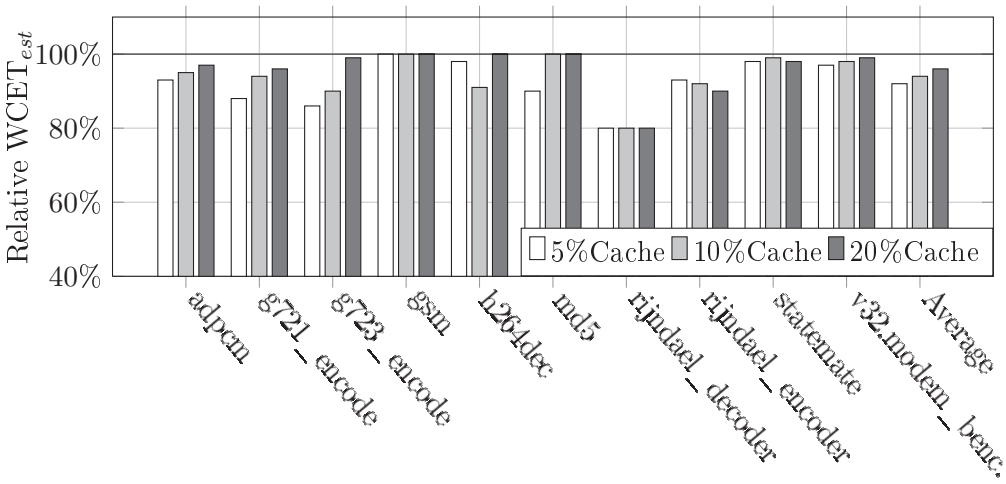


Figure 3.11: Relative WCET$_{est}$s after memory content selection

not be achieved. Therefore, the cache sizes were artificially limited to 5, 10 and 20% of the program's overall code size. This guarantees that a similar ratio of cache size to program size is used for all optimizations and static WCET analyses, found in current embedded systems in order to generate comparable results.

3.4.4.1 WCET Estimations

Figure 3.11 depicts the results achieved by our new memory content selection algorithm for the considered 10 benchmarks. For each benchmark, each of the three bars represents the results for different cache sizes. The bars depict the estimated WCET of the optimized program computed by the static WCET analyzer relative to the estimated WCET if the benchmark is executed with all functions located in a cached memory region.

Our algorithm was able to reduce the estimated WCET of the benchmarks by up to 19.5% for 5% and 10% cache size for the *rijndael_decoder*. For the same benchmark, the proposed optimization was able to achieve a WCET$_{est}$ reduction of 20.1% for 20% cache size. The reasons why the worst-case performance of benchmark *gsm* could not be improved are twofold. First, the loop of a filter function which consumes about 95% of the program's estimated WCET amounts to only 3% of the program size and fits into all considered cache sizes. Second, the program's control flow is highly serial with only few branches. Thus, only few cache conflict misses occur and the best performance is achieved if all functions reside in a cached memory area during execution.

Due to the fact that our optimization is based on a heuristic to select the most promising functions, wrong decisions could be made leading to suboptimal results. The benchmark *h264dec* clearly indicates suboptimal results for the cache sizes of 5% and 20% of the program size. For these cases, the memory content selection

algorithm optimized on a local minimum where a different set of functions allocated to the cached memory would achieve a better worst-case performance.

On average, the estimated WCET of all benchmarks could be reduced by 8% for the smallest considered cache size and by 6 and 4% for 10% and 20% cache size, respectively.

For most of the benchmarks, it can be observed that our algorithm performs better for smaller cache sizes which has two reasons. On the one hand, there is more optimization potential since the number of cache misses necessarily grows with caches diminishing in size. On the other hand, the *principle of locality* [25] which states that programs tend to reuse data or instructions they have used recently also applies to most of the programs running on embedded systems. Based on the principle of locality, a widely known rule of thumb is that most of the programs spend 90% of their execution time in only 10% of the code [51, p. 47]. As a consequence, the considered 10% cache size can often already keep copies of a program's hot spots. Thus, it is very likely that the execution time of the program can be improved by at most 10% by storing the remaining 90% of the code in the cache. The middle bar of each benchmark shown in Figure 3.11 represents exactly these 10% cache size. The 100% line represents the WCET_{est} of a system with a normally operating cache which can store copies of just these program's hot spots. Seen from this perspective, the achieved average WCET_{est} reductions of 4-8% seem to be hardly improvable.

3.4.4.2 Optimization Time

To consider the optimization time, an Intel Xeon X3220 (2.40 GHz) was utilized. Most of the time necessary for our novel WCET-driven cache-aware memory content selection algorithm was consumed by the repetitive WCET analyses using *aiT*. For a program consisting of n functions, the maximum number of WCET analyses amounts to:

$$\#Analyses_{WCET} = 3 + n \tag{3.20}$$

Initially, two analyses have to be performed in order to calculate the profit (line 4) and one for determining the reference WCET (line 15). Another analysis is required after each allocation of a function to evaluate its impact on the overall WCET (line 19).

Most of the time was consumed by the optimization of the *rijndael_decoder* where 6 WCET analyses require almost 2 hours of CPU time. The highest number of WCET analyses (17) was performed during the optimization of the two benchmarks *g721_encode* and *g723_encode* which amounts to 8 and 10 minutes of analysis time, respectively.

The optimization presented in this section lead to publication [96].

3.5 Summary

This chapter presented novel WCET-driven memory-based optimization techniques which optimize the assembly code of a program. All optimizations are integrated into the WCC compiler framework presented in Chapter 2. Unlike prevalent ACET optimizations which are based on heuristics or exploit profiling information to apply code modifications, the presented optimizations rely on WCC's integrated WCET timing model. Based on this WCET data, optimizations are able to focus on the optimization of those code parts lying on the WCEP. Thereby, the impact of applied code transformations on the WCET can easily be evaluated to prevent adverse effects. The effectiveness of the discussed optimizations exploiting processor-specific features was demonstrated on real-life benchmarks.

The first optimization proposed in this chapter was branch prediction aware code positioning with the objective to decrease the WCET of a program. The order of basic blocks inside a function is changed with the goal to support the branch prediction of conditional branches in order to fetch the most frequently visited successor in advance and to avoid unconditional branches. An evolutionary approach is presented which employs the well-known techniques mutation and crossover. Offspring individuals are created whose order of basic blocks hopefully tends to decrease the WCET of the represented program. An ILP-based approach was introduced avoiding time-consuming repetitive WCET analyses. Therefore, the control flow of a program and the resulting jump penalties were explicitly modeled to determine an improved order of basic blocks w. r. t. the WCET. By applying these techniques, a WCET decrease of up to 24.7% can be achieved. The evolutionary approach was able to reduce the average WCET reductions by 8.9% with the drawback of repetitive WCET estimations incurring high optimization times. The ILP-based algorithm was able to decrease WCET of programs by 6.7% on average.

Secondly, a novel algorithm for WCET-driven cache-aware memory content selection was introduced. It was shown how the I-cache performance can be improved in order to decrease the WCET of a program. Therefore, the presented greedy algorithm selects the set of functions to be cached. Only frequently used functions whose WCET highly benefits from a cached execution are moved to cached memory regions. This ensures that they cannot be evicted from the cache by functions with a lower WCET decrease being allocated to non-cached memories. The influence on the WCET of a program is evaluated when a promising function is moved to a cached memory region in order to always optimize along the WCEP. Applying this technique, a WCET decrease of up to 20% can be achieved, and at the same time, it can be ensured that the performance of the optimized program is never worse than the original. On average, WCET reductions of 4% to 8% were achieved for cache sizes ranging from 5% to 20% of the overall code size.

The results presented in this chapter indicate that optimizations can have adverse effects on different objectives. In order to optimize a certain objective, this objective has to be considered during the code transformation process so that the

full optimization potential can be exploited. In case of WCET optimization, the resulting WCET data of a program has to be taken into account. Profiling-based approaches cannot guarantee good or even optimal performance as well since a frequently used path of a program, identified as critical path, does not necessarily be equal to the WCEP. However, considering and shortening the WCEP, as done by the optimizations presented in this chapter, immediately decreases the WCET – possibly by worsening concurrent paths not contributing to the WCET. For input data not leading to the WCET, such concurrent paths can yet be frequently used paths for which a worsening leads to an increased ACET.

Furthermore, this chapter shows that only optimizing compilers considering the underlying hardware platform can yield the best performance by exploiting specialized hardware features. Especially the employed memory system has to be taken into account since exploiting available memories and optimizing access patterns of programs shows a high optimization potential. Otherwise, units such as caches can have an adverse effect on the WCET and predictability of a system.

Chapter 4

Optimization of Multi-Task Systems

Contents

4.1 Introduction . . . 72
4.2 Existing Multi-Task Approaches . . . 73
4.3 Multi-Task Compiler Extensions . . . 74
4.4 WCET-aware Software Based Cache Partitioning for Multi-Task Systems . . . 76
4.4.1 Motivating Example . . . 77
4.4.2 Related Work . . . 78
4.4.3 Software-based Cache Partitioning . . . 79
4.4.4 Size-driven Partition Size Selection . . . 82
4.4.5 WCET-driven Partition Size Selection . . . 83
4.4.6 Evaluation . . . 85
4.4.6.1 WCET Estimations . . . 86
4.4.6.2 Optimization Time . . . 88
4.5 WCET-driven Multi-Task Program Scratchpad Allocation 88
4.5.1 Motivating Example . . . 89
4.5.2 Related Work . . . 90
4.5.3 Multi-Task Program Scratchpad Allocation . . . 91
4.5.4 Evaluation . . . 93
4.5.4.1 WCET Estimations . . . 94
4.5.4.2 Optimization Time . . . 96
4.6 Memory Architecture aware Compilation . . . 97
4.6.1 Motivating Example . . . 98
4.6.2 Related Work . . . 99
4.6.3 Optimization Sequence Determination . . . 99
4.6.4 Modification of Optimizations . . . 100
4.6.5 Evaluation . . . 103
4.6.5.1 WCET Estimations for Cache Partitioning combined with Memory Content Selection . . . 104
4.6.5.2 WCET Estimations for Multi-Task SPM Allocation combined with Cache Partitioning . . . 105

4.6.5.3 WCET Estimations for Multi-Task SPM Allocation, Cache Partitioning and Memory Content Selection 107
4.6.5.4 Optimization Time 108
4.7 Summary 110

4.1 Introduction

In the course of this thesis, Chapters 2 and 3 presented the WCC compiler framework and how it can be employed to develop optimizations aiming at WCET reduction of single programs. Current embedded/cyber-physical systems, however, exhibit the computation power to execute a number of concurrent tasks. Since the demands on such systems are growing as well, multi-task systems are employed which often implement various functional units in a single system. Existing compilation frameworks lack a multi-task representation and therefore are not suited for the optimization of such systems: If, for instance, tasks do not meet their deadlines, a standard compiler is not able to prioritize those tasks during optimization. A multi-task aware compiler, in contrast, could assign scratchpad memory to such tasks in order to speed up their execution with the result that all deadlines are (hopefully) met.

In this chapter, novel compiler extensions and optimizations for systems running multiple tasks are presented. All optimizations focus on the reduction of the overall WCET_{est} of a set of programs running on the TriCore TC1796/TC1797 presented in Section 2.5. Section 4.2 gives an overview on existing multi-task compiler framework before Section 4.3 proposes extensions which make the WCC aware of multi-task systems. Afterwards, Section 4.4 introduces a software-based cache partitioning for multi-task systems with the objective of WCET reduction. In order to introduce predictability, the cache is partitioned and each partition is exclusively assigned to a certain task. This ensures that tasks do not compete for the same cache lines and evict each other from the cache. An ILP is employed to select the optimal partition size w. r. t. a system's overall WCET_{est}. As second optimization, a technique for extending single-task scratchpad memory allocation techniques to multi-task scenarios is proposed in Section 4.5. Three different heuristics are presented which determine the amount of scratchpad memory assigned to each task in order to decrease the overall system's WCET. An established scratchpad allocation technique for single tasks is exploited to perform an allocation for each task and the determined scratchpad size.

Up to now, at most one single WCET-driven optimization is applied to a program to optimize and nothing is known about possible interactions or side-effects of optimization chains. Thus, Section 4.6 tackles this issue by evaluating the effects of combined WCET-driven optimizations. A strategy for an intelligent cooperation of

multi-task scratchpad memory allocation, cache partitioning and cache-aware memory content selection is presented. Finally, Section 4.7 provides a brief summary and concludes this chapter.

4.2 Existing Multi-Task Approaches

Recent literature covering the design process of embedded/cyber-physical systems also considers systems running multiple tasks in parallel. In [82], Marwedel describes the design process of real-time systems including available real-time operating systems and basics of real-time task scheduling. Compilation and optimization techniques for embedded systems are presented as well, but tasks are always considered separately.

Gebhard et al. try to improve the cache performance by applying task placement techniques [42]. Therefore, the interdependency relation is evaluated in order to maximize the number of persistent cache sets for each task by an optimized memory layout. Although multi-task sets are optimized, the presented approach is not integrated into a compiler but expects precompiled object files for which a suitable linker script is generated which reflects the optimization decisions.

Up to now, no fully integrated compiler framework exists which supports a system developer in creating multi-task software projects for embedded/cyber-physical systems. Especially the integration of a static timing analyzer and multi-task capabilities are missing in existing compilers. This makes an automated compilation, optimization and analysis workflow infeasible.

WCET timing analysis of multi-task systems is more complicated than the analysis for a single task system. *aiT*, for instance, is only able to analyze a single program under the assumption that the task is not interrupted by another task, scheduler or interrupt handlers. This is due to the fact that the point of time for such an interruption is not predictable. Thus, the program point, the resulting pipeline and cache states at which an interruption occurs cannot be determined. Each time a timing analyzer is not able to determine the actual behavior, the worst case has to be assumed. This would be interruptions at arbitrary points of time in a multi-task scenario. For each of these points, an analysis has to flush the processor pipeline and invalidate the cache content. The result would be a highly overestimated WCET.

Even if the points are arbitrary at which a program is interrupted in a multi-task scenario, the number of interruptions is usually limited. Such preemptions of tasks mainly lead to additional costs caused by cache misses if different tasks are mapped to the same cache blocks. This fact is exploited by Lee et al. who introduced a *cache-related preemption delay* (CRPD) analysis [66]. A path analysis of the preempted tasks is performed to determine the program point with the highest cache utilization. Based on the number of utilized cache sets, the CRPD for a single preemption can be computed. Considering the number of possible preemptions, an upper bound of the task's WCET including CRPD can be determined.

The concept of CRPD usually overestimates the number of required cache refills caused by preemption. Not all cache blocks of a preempted task will be reused and a preempting task does not evict all content from the cache. Therefore, the notion of *useful cache blocks* (UCB) and *evicting cache blocks* has been introduced by Burguière et al. [15]. Considering UCB and ECB during cache analysis can reduce the overestimation for direct-mapped caches and way-associative caches with LRU replacement policy.

For set-associative caches, an access of a preempting task to a cache set does not necessarily lead to an eviction of a preempted task. Altmeyer et al. tighten the CRPD analysis for set-associative caches with LRU replacement policy by computing the *resilience* of memory blocks of the preempted task [4]. The *resilience* of a memory block is the number of accesses which can be performed to the same cache set without evicting the block. Considering the *resilience* during multi-task cache analysis helps to reduce the overestimation compared to the UCB/ECB approach.

Kleinsorge et al. combine the most accurate algorithms for CRPD, UCD and ECB analysis for set-associative caches [63]. Compared to existing approaches which either exhibits high accuracy or high performance at the cost of higher overestimation, improvements are presented which enable short analysis times at highest possible accuracy.

All presented approaches considering the analysis of multi-task systems have in common that they introduce a certain amount of overestimation since the analyzed systems are not fully predictable. To overcome this obstacle, this chapter presents optimization techniques which make multi-task systems fully predictable.

4.3 Multi-Task Compiler Extensions

The WCC compiler framework presented in Chapter 2 is eminently suited for the compilation and optimization of single task programs w. r. t. the resulting WCET_{est}. In order to extend the existing compilation workflow to handle multi-task systems, a multi-task representation was added to the WCC compiler. In the following, the necessary modifications are presented.

According to Figure 2.2, p. 16, the single-task WCC reads one or more source files belonging to a single program to optimize. These source files are represented by the high-level IR *ICD-C* (cf. Section 2.4.1) and a semantically equivalent sequence of assembly instructions maintained in a list of *LLIR*s (cf. Section 2.4.3). Such a representation is insufficient for maintaining a set of tasks with their scheduling parameters, possibly a task-specific compiler configuration and the ICD-C as well as the LLIRs representing the task itself. Hence, the WCC was extended by a multi-task representation called *task set* which can comprise a number of tasks in the form of *task entries*. Figure 4.1 depicts the resulting optimization flow.

In contrast to WCC's original workflow, the compiler now expects the specification of a *task set* as input. Based on this specification, a number of *task entries* is

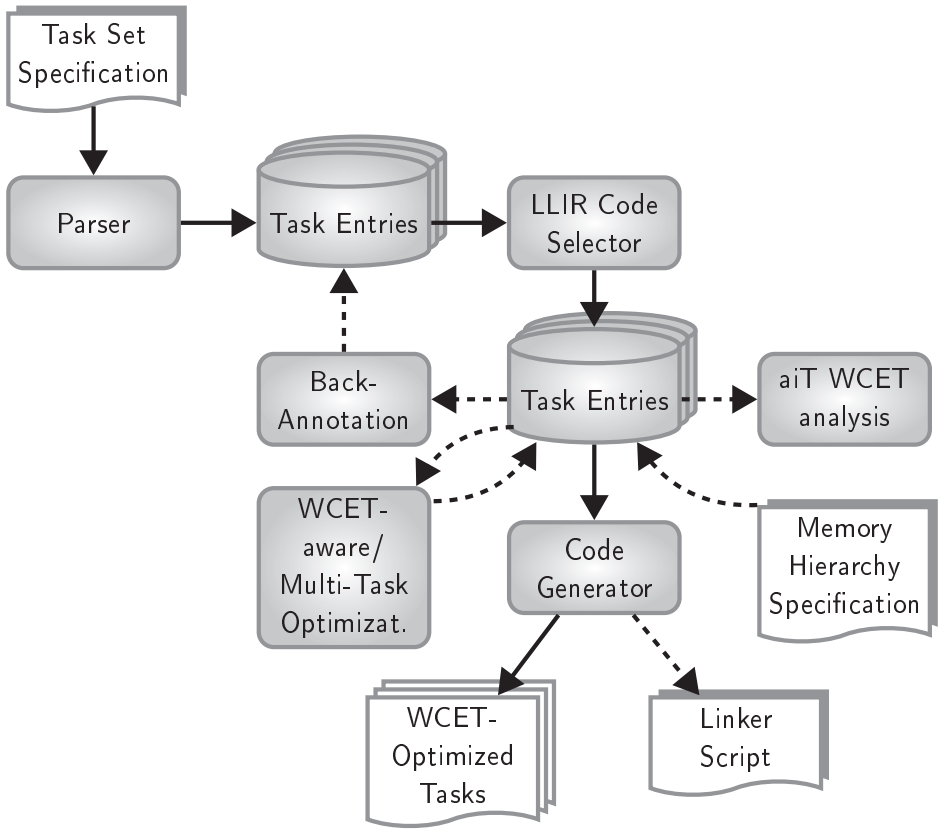


Figure 4.1: Workflow of the WCET-aware C compiler WCC with multi-task extensions

created. Compiling and optimizing the *task set* is performed for each contained task separately. Since WCC's internal memory layout reflects the target architecture, all tasks share the same address space. Thus, modifying the code or data layout of a single task can have an impact on the memory layout of all remaining tasks in a *task set*. This ensures that WCET estimations performed on single tasks always implicitly consider the resulting memory layout of the final multi-task system. Otherwise, safe guarantees concerning the runtime of individual tasks and all the more of the entire system cannot be made.

In a final optimization step, dedicated multi-task optimizations are applied to the entire *task set* as presented in the remainder of this chapter.

Specification of Task Sets

Compared to standard compilers, only specifying a list of ANSI-C source files is not sufficient for multi-task compilation since it cannot be distinguished to which task a source file belongs. Hence, an XML-based file format for specifying task sets

```
<task>
  <name>mult_4_4</name>
  <sources>
    <file>UTDSP/mult_4_4/mult_4_4.c</file>
    <file>UTDSP/mult_4_4/input_dsp.c</file>
  </sources>
  <scheduling>
    <period>1000000</period>
  </scheduling>
  <wccrc>UTDSP/mult_4_4/.wccrc</wccrc>
</task>
```

Listing 4.1: Exemplary task specification file

was introduced which is read by WCC as input. An example for the employed file format is depicted in Listing 4.1.

The described task set can comprise multiple tasks enclosed by `<task>` tags which are identified by a unique name in `<name>` tags. For each task, a list of source files (between `<sources>` tags) can be specified – each initiated by a `<file>` tag. If scheduling of tasks should be considered, scheduling parameters can be added – such as the `<period>` in processor cycles or the task's priority. A separate compiler configuration file can be specified by `wccrc` tags which can contain miscellaneous parameters such as parameters steering the analysis precision of the WCET analysis. Task-individual optimization parameters could be specified in this file as well.

Task Set Representation

If WCC parses a task set specification, a data structure named *task set* is created which contains a number of *task entries* representing the specified tasks running on a system. According to Figure 4.2, a global memory layout is an inherent part of a *task set* reflecting the system's memory layout based on the present tasks. An ICD-C IR which is generated from the task's source files can be attached to each *task entry*. The list of LLIRs, generated during WCC's code selection phase (cf. Section 2.4.2, p. 18), is attached to the corresponding *task entry* as well. In this way, a *task entry* holds all data structures in different abstraction levels which represent a task running on a system.

4.4 WCET-aware Software Based Cache Partitioning for Multi-Task Systems

Nowadays, embedded systems have to fulfill a growing number of functions and thus, the number of tasks running on such a system is growing as well. This section

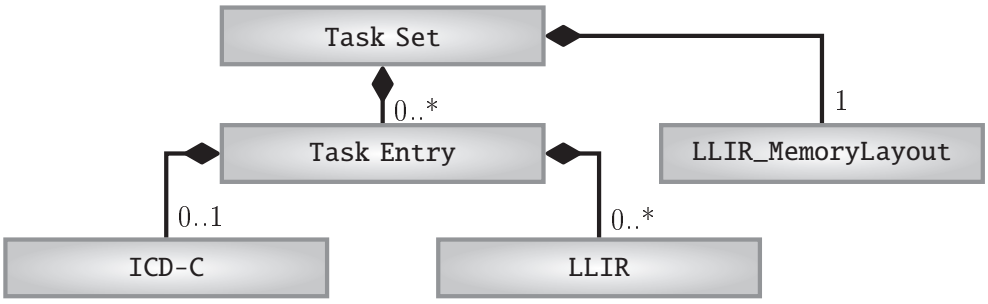


Figure 4.2: Class hierarchy of multi-task representation

presents a cache partitioning approach which is able to optimize such multi-task systems by dividing the cache into disjoint partitions. Optimization techniques can assign code or data objects to these partitions in order to ensure that objects from different partitions cannot evict each other. Cache partitioning can be performed either in hardware where the cache controller logic ensures a proper mapping of tasks to their partitions. Or cache partitioning can be realized in software by exploiting the modulo addressing function of the cache controller.

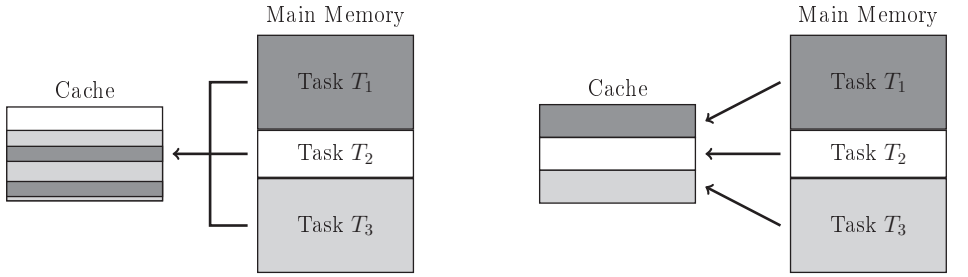
Since no commercial available CPU equipped with caches integrates a hardware-based partitioning scheme, this chapter introduces software-based cache partitioning techniques. An existing technique [89] is adapted to make the instruction cache (*I-cache*) behavior more predictable. This can be guaranteed since every task has its own cache partition from which it cannot be evicted by another task. A novel WCET-aware cache partitioning aims at selecting the optimal partition size for each task of a set to achieve the minimal overall WCET.

The rest of this chapter is organized as follows: The next section illustrates the problems of multi-task systems equipped with caches in order to motivate the techniques presented in this chapter. Section 4.4.2 gives an overview of related work. Existing techniques to partition a cache as well as our new algorithm are explained in Sections 4.4.3 – 4.4.5. An evaluation of the performance which is achieved by our WCET-aware cache partitioning is presented in Section 4.4.6.

4.4.1 Motivating Example

In environments with preemptive schedulers running more than one task, it is often impossible to make proven assumptions about memory access patterns. This is mainly caused by interrupt-driven scheduling algorithms causing context switches at unknown points of time. Figure 4.3a shows a possible memory layout and a resulting cache state after a number of task switches. Assuming that task T_1 initially occupied the complete cache and tasks T_2 as well as T_3 can interrupt the execution of task T_1, parts of T_1 are evicted by the other tasks.

Since the program point is not known at which a context switch occurs, it is also unknown at which address the execution of e.g. task T_1 continues. Hence,



(a) Cache thrashing in preemptive scheduled systems

(b) Mapping of tasks to partitioned cache

Figure 4.3: Comparison between normally operating and partitioned cache

it is usually unpredictable which line of the cache is evicted next. An unknown cache behavior forces a timing analyzer to assume a cache miss for every memory access implying a highly overestimated system's overall WCET. In order to reduce this overestimation, Section 4.2 presented various cache analysis techniques which help to narrow the number of cache block possibly evicted due to a context switch. Since an overestimation cannot entirely be avoided, the underlying system has to be oversized to meet real-time constraints resulting in higher costs for hardware.

To overcome this situation, partitioned caches are recommended in literature [89, 22, 87] which are divided into disjoint partitions. Each of these partitions is assigned to exactly one task. As depicted in Figure 4.3b, tasks then can only evict cache lines residing in the partition they are assigned to. Reducing the prediction problem of cache interference between tasks to one task with its own cache partition makes the system fully predictable w. r. t. context switches. Since tasks cannot evict each other, no *CRPD* occurs and the context switch overhead is reduced to refilling the processor pipeline with instructions of the preempted task after the preempting task is finished. Thereby, an overestimation of the *CRPD* can be entirely avoided.

Such a configuration allows the application of well-known single-task approaches for WCET- and cache performance estimation. The overall execution time of a task set is then composed of the execution time of the single tasks with a certain partition size and the overhead required for scheduling including additional time for refilling the processor pipeline.

4.4.2 Related Work

The papers [89, 22, 87] present different techniques to exploit cache partitioning realized either in hardware or in software. In contrast to our work, these implementations either do not take the impact on the WCET into account or do not employ the WCET as the key metric for optimization which leads to suboptimal or even degraded results. In [89], the author presents ideas for compiler support for software-based cache partitioning which serves as basis for the partitioning tech-

niques presented in this paper. Non-linear control flow transformations are proposed for instruction cache partitioning whereas partitioning for data caches involves code transformation of data references. Compared to the work in this paper, a functional implementation or impacts on the WCET are not shown.

In [22], a hardware extension for caches is proposed to realize a dynamic partitioning through a fine-grained control of the replacement policy via software. Access to the cache can be restricted to a subset of the target cache set which is called columnization. For homogeneous on-chip multi-processor systems sharing a unified set-associative cache, [87] presents partitioning schemes based on associativity and sets.

A combination of locking and partitioning of shared caches on multi-core architectures is researched in [106] to guarantee a predictable system behavior. Even though the authors evaluate the impact of their caching schemes on the worst-case application performance, their algorithms are not WCET-aware. Kim et al. [60] developed an energy-efficient partitioned cache architecture to reduce the energy dissipation per access. A partitioned L1 cache is used to access only one sub-cache for every instruction fetch leading to dynamic energy reduction since other sub-caches are not accessed.

The authors of [21] show the implications of code-expanding optimizations on instruction cache design. Different types of optimizations and their influence on different cache sizes are evaluated. [64] gives an overview of cache optimization techniques and cache-aware numerical algorithms. The authors focus on the memory interface which often limits the performance of numerical algorithms and point out ways for the development of cache-aware algorithms to overcome this issue.

Puaut et al. counteract the problem of unpredictability with locked instruction caches in multi-task real-time systems. They propose two low-complexity algorithms for cache content selection in [99]. A drawback of statically locking the cache content is that the dynamic behavior of the cache gets lost. Code is no more automatically loaded into the cache, thus code which is not locked into the cache cannot profit from it anymore.

Vera et al. [117] combine cache partitioning, dynamic cache locking and static cache analysis of data caches to achieve predictability in preemptive systems. This eliminates overestimation and allows to approximate the worst-case memory performance.

4.4.3 Software-based Cache Partitioning

The author of [89] presents a theory to integrate software based cache partitioning into a compiler toolchain without an existing implementation. Code should be scattered over the address space so that tasks are mapped to certain cache lines. Therefore, all tasks have to be linked together in one monolithic binary, and a possible free space between several parts has to be filled with *NOPs*. Partitioning for data caches involves code transformation of data references.

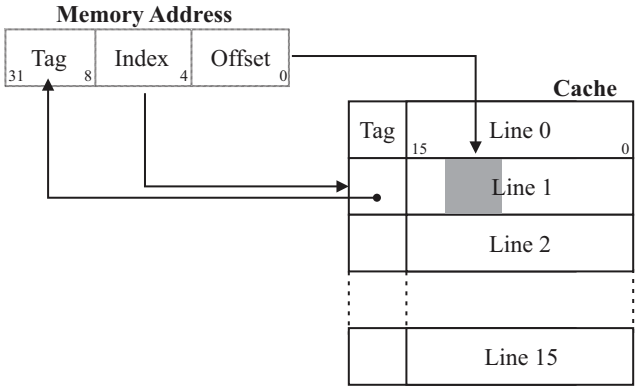


Figure 4.4: Addressing of cache content

The theory to exactly position code in the address space to map it into certain cache lines is picked up here, but a completely different technique is applied to achieve such a distribution. We restrict ourselves to partitioning of I-caches, thus only software based partitioning of code using the new technique is discussed. However, a partitioning of data caches w.r.t. WCET decrease is straightforward using a modified version of our algorithm.

For the sake of simplicity, a direct-mapped cache is assumed in the following. Way-associative caches can be partitioned as well: the desired partition size has to be divided by the degree d of associativity since any particular address in main memory can be mapped in one of d locations in the cache. The replacement policy has no influence on the partitioning procedure since a replacement only happens within a task's partition.

Assuming a very small cache with $S = 256$ bytes capacity divided into cache lines of $S_{line} = 16$ bytes size. When an access to a cached memory address is performed, the address is split into a tag, an index, and an offset part. For a system with byte-based addressing, the example in Figure 4.4 shows the $\mathit{Off} = log_2(S_{line}) = 4$ offset bits addressing the content inside a cache line, whereas $\mathit{Ind} = log_2(S/S_{line}) = log_2(16) = 4$ index bits select one of $l = 2^{\mathit{Ind}} = 16$ cache lines. The remaining address bits form the tag which is stored in conjunction with the cache line. The tag bits have to be compared for every cache access since arbitrary memory addresses with the same index bits can be loaded into the same line.

The cache line a memory address is mapped to is calculated as follows:

$$\mathit{line}(\mathit{Addr}) = (\mathit{Addr} \gg \mathit{Off}) \bmod l \tag{4.1}$$

For a memory address Addr, the least significant Off bits addressing a word within a cache line are stripped by shifting right the required number of bits. The addressed cache line is determined by computing the modulus of the remaining relevant bits and the number of cache lines l.

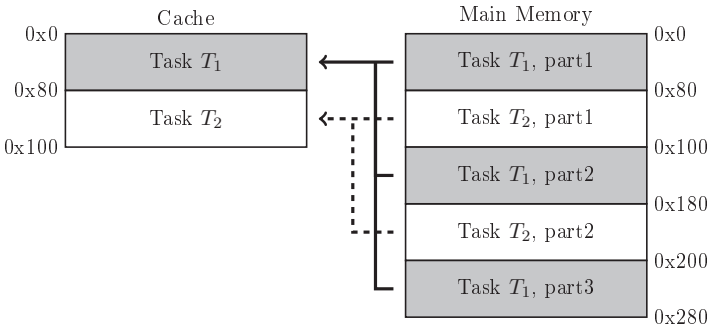


Figure 4.5: Mapping of tasks to cache lines

To partition a cache, it has to be ensured that a task assigned to a certain partition only allocates memory addresses mapped to cache lines belonging to this partition. For an instruction cache divided into two partitions of 128 bytes, one partition ranges from cache line 0 to line 7 and the second one from line 8 up to 15. If a task T_1 is assigned to the first partition, each occupied memory address must have index bits ranging from `0000b` up to `0111b` accessing the cache lines 0 to 7. According to Equation 4.1, the least significant bits of an address corresponding to index and offset bits ranging from `0x00` to `0x7f` are mapped to the desired cache lines 0 up to 7. A task T_2 assigned to the second partition has to be restricted to cover only memory addresses with the least significant bits ranging from `0x80` up to `0xff` (index bits `1000b` up to `1111b`).

Tasks exceeding the size of the partition they are mapped to have to be split and scattered over the address space. Figure 4.5 illustrates the partitioning for tasks T_1 and T_2 into such 128 bytes portions and the distribution of these portions over the main memory. Task T_1 is allocated to portions which are mapped to the first half of the cache since all occupied memory addresses modulo cache size range from 0 to 127. The same has to meet for task T_2 occupying memory addresses modulo cache size ranging from 128 to 255.

Obviously, partitioning does not depend on the cache line size since a contiguous part of the memory is always mapped into the same amount of cache memory. Only the atomic size for composing partitions is equal to the cache line size, thus the partition size must be a multiple thereof.

In contrast to the approach in [89], WCC's workflow employs the linker to achieve such a distribution of code over the address space. Individual linker scripts are used (compare Listing 4.2 for the aforementioned example) with relocation information for every task and its portions it is divided into. For linker basics refer to [19].

The output section `.text` to be created in the output binary (line 1), is aligned to a memory address which is a multiple of the cache size to ensure that the mapping starts at cache line 0. Line 3 allocates the assembly input section `.task1_part1` at the beginning of the `.text` output section, thus the beginning of the cache. The content of this section must not exceed 128 bytes since line 4 sets the relocation

```
1  .text: {
2      _text_begin = .;
3      *(.task1_part1)
4      . = _text_begin + 0x80;
5      *(.task2_part1)
6      . = _text_begin + 0x100;
7      *(.task1_part2)
8      . = _text_begin + 0x180;
9      *(.task2_part2)
10     . = _text_begin + 0x280;
11     *(.task2_part3)
12 } > PFLASH-C
```

Listing 4.2: Linker script example

counter to the address 128 bytes beyond the start address, which is mapped into the first line of the second cache half. Line 5 accomplishes the relocation of section `.task2_part1` to the new address. The other sections are mapped in the same manner.

On assembly level, each code portion which should be mapped to a partition, has to be attached to its own linker section to cause a relocation by the linker; e.g. `.task_part1` for the first 128 bytes memory partition of task T_1. To restore the original control flow, every memory partition has to be terminated by an additional unconditional branch to the next memory partition of the task unless the last instruction of this block already performs an unconditional transfer of control.

For this and further jump corrections required by growing displacements of jump targets and jump sources, WCC's integrated jump correction is applied (refer to Section 2.4.7, p. 26).

4.4.4 Size-driven Partition Size Selection

The author in [89] proposes to select a task T_i's partition size depending on its size relative to the size of the complete task set. Since the original version does not take scheduling into account, the approach is extended to consider the execution frequency f_i of T_i in a hyperperiod H.

Definition 4.1 (*Hyperperiod*)

In systems with a number of n periodically scheduled tasks, the timespan after which the schedule repeats is called *hyperperiod*. Since each task T_i is invoked an integer number of times, the length of the hyperperiod H is the least common multiple of the tasks' periods p_i:

$$length(H) = lcm(p_1 \dots p_n)$$

As a consequence, task T_i is executed f_i times within the period H:

$$f_i = \frac{length(H)}{p_i}$$

The hyperperiod and the tasks' execution count can be computed based on the values provided by WCC's multi-task representation (cf. Section 4.3). Based on T_i's code size $s(T_i)$, its partition size computes as follows:

$$p(T_i) = \frac{f_i * s(T_i)}{\sum_{j=1}^{|T|} f_j * s(T_j)} * S_{cache} \tag{4.2}$$

Example 4.1

A task set with $|T| = 4$ tasks $T_1 - T_4$ with the following code sizes $s(T_i)$ and execution counts f_i in a hyperperiod is assumed:

Task	$s(T_i)$	f_i
T_1	*32*	*4*
T_2	*128*	*2*
T_3	*512*	*1*
T_4	*128*	*1*

The complete task set has an overall code size of 800 bytes, whereas the assumed cache from the previous section with a capacity of $S = 256$ bytes is used. According to Equation (4.2), T_1 is assigned to a partition with the following size:

$$\begin{aligned} p(T_1) &= \frac{4 * 32\,bytes}{4 * 32\ bytes + 2 * 128\,bytes + 512\,bytes + 128\,bytes} * 256\,bytes \\ &= \frac{128}{1024} * 256\,bytes \\ &= 1/8 * 256\,bytes \\ &= 32\,bytes \end{aligned}$$

Thus, the assigned partition sizes are: $p(T_1) = 32$ bytes, $p(T_2) = 64$ bytes, $p(T_3) = 128$ bytes and $p(T_4) = 32$ bytes.

4.4.5 WCET-driven Partition Size Selection

The size of a cache may have a drastic influence on the performance of a task or an embedded system. Caches with sufficient size can decrease the runtime of a program whereas undersized caches can lead to a degraded performance due to a high cache miss ratio. Hence, it is essential to choose the optimal partition size for every task in order to achieve the highest possible decrease of the system's overall WCET.

Current approaches select the partition size depending on the code size or a task's priority [89, 106]. They aim at improving a system's predictability and examine the impact of partitioning on the WCET but do not explicit aim at minimizing its WCET.

In this section, a novel approach is presented to determine the optimal partition sizes for a set of tasks w.r.t. the lowest overall WCET of a system. *Integer-linear programming* is utilized to select the partition size for each task from a given set of possible partition sizes.

As in Section 4.4.4, a set T comprising $|T|$ tasks which are scheduled periodically is assumed. Furthermore, a set P of given partition sizes is assumed with $|P|$ partitions, e.g. $P = \{0, 128, 256, 512, 1024\}$ measured in bytes. Let x_{ij} be a binary decision variable determining if task T_i is assigned to a partition with size $P_j \in P$:

$$x_{ij} = \begin{cases} 1, & \text{if } T_i \text{ assigned to } P_j \\ 0, & \text{else} \end{cases}$$

To ensure that a task is assigned to exactly one partition, the following $|T|$ constraints have to be met:

$$\forall i = 1..|T| : \sum_{j=1}^{|P|} x_{ij} = 1 \tag{4.3}$$

$WCET_{ij}$ is T_i's WCET for a single execution if assigned to partition P_j; the WCET for a single task T_i is calculated as follows:

$$w(T_i) = \sum_{j=1}^{|P|} x_{ij} * WCET_{ij} \tag{4.4}$$

Focused on WCET minimization, a cost function modeling the system's overall WCET based on the tasks' WCET and execution frequency within a hyperperiod is defined:

$$WCET = \sum_{i=1}^{|T|} f_i * w(T_i) \rightsquigarrow min. \tag{4.5}$$

Although scheduling requires a certain amount of computational power and thereby has influence on the systems overall WCET, the number of context switches and the resulting overhead remains constant. Thus, the scheduling overhead has no influence on the optimization decisions and is omitted in Equation (4.5).

To keep track of the limited cache size S, another constraint is introduced:

$$\sum_{i=1}^{|T|} \sum_{j=1}^{|P|} x_{ij} * P_j \ \leq S \tag{4.6}$$

Using equations (4.3) to (4.6), the cost function and $m+1$ constraints can be set up as input for an ILP solver like *lp_solve* [79] or *CPLEX* [56]. After solving the set

of linear inequations, the minimized WCET and all variables $x_{ij} = 1$, representing the optimal partition sizes for all tasks, are known.

To determine the constant values $WCET_{ij}$ in order to set up all equations and to apply partitioning, Algorithm 2 is employed. A given task set, the instruction cache size, a set of possible partition sizes as well as execution counts in a hyper period of the tasks are required as input data. The algorithm iterates over all tasks (line 2) and temporarily partitions each task (line 3 to 4) for all given partition sizes. Subsequently, the WCET for the partitioned task is determined invoking AbsInt's static analyzer *aiT* (line 5). Exploiting the information about tasks, their execution counts, partition sizes, cache size and gathered WCETs, an ILP model is generated regarding equations (4.3) to (4.6) (line 9) and solved in line 10.

Algorithm 2 Pseudo code of cache partitioning algorithm

Require: Set of tasks T, set of partition sizes P, execution counts C, cache size S

```
 1: set<WCET> WCETs = ∅
 2: for T_i ∈ T do
 3:    for P_j ∈ P do
 4:       partitionTaskTemporary( T_i, P_j )
 5:       WCET WCET_ij = determineWCET( T_i )
 6:       WCETs = WCETs ∪ WCET_ij
 7:    end for
 8: end for
 9: ilp = setupEquations( T, P, WCETs, C, S )
10: X = solveILP( ilp )
11: for all x_ij ∈ X : x_ij = 1 do
12:    partitionTask( T_i, P_j )
13: end for
14: return T
```

Afterwards, the set X includes exactly one decision variable x_{ij} per task T_i with the value 1 whereas P_j is T_i's optimal partition size w.r.t. minimization of the system's WCET. Finally, lines 11 to 12 perform software based partitioning of each task with its optimal partition size, as described in Section 4.4.3.

4.4.6 Evaluation

This section evaluates the performance of our WCET-driven cache partitioning and compares it to existing partition size selection heuristics based on tasks sizes. Different task sets from media and real-time benchmark suites are used to evaluate our optimization on computing algorithms typically found in the embedded systems domain. Namely, tasks from the suites *DSPstone* [130], *MRTC* [46] and *UTDSP* [115] are evaluated. WCC's support for the Infineon TriCore architecture in the form of the TC1796 processor is employed for the evaluation. The processor integrates a 16 kB 2-way set associative I-cache with 32 bytes cache line size. For the

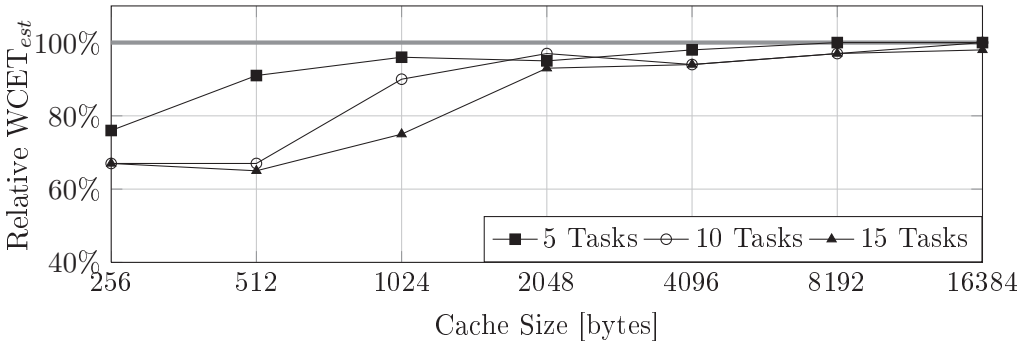


Figure 4.6: Optimized WCET$_{est}$ for DSPstone relative to standard approach

evaluation, seven cache sizes with the power of two are taken into account, ranging from 256 bytes up to 16 kB.

Overall, the used benchmark suites include 101 benchmarks so that we have to limit the evaluation to a subset of tasks for cache partitioning. Due to the lack of specialized multi-task benchmarks suites, sets of tasks stemming from the aforementioned benchmark suites, as proposed in [50], are generated and compiled with the optimization level *O3*. Using these sets, the capability of decreasing the WCET achieved by standard partitioning algorithms compared to our WCET-aware approach is determined.

Different numbers of tasks (5, 10, 15) in a set are checked to evaluate their effect on the WCET. To gather reliable results, the average of 100 sets of randomly selected tasks is computed for each considered cache size and task set size. The overall number of ILPs for every benchmark suite, which has to be generated and solved for the seven considered cache sizes, is:

$$|ILPs| = 3 * 100 * 7 = 2100$$

Equation (4.5) modeling the systems overall WCET takes scheduling into account. Since this evaluation employs synthetic task sets for benchmarking, the task's execution frequencies f_i in this equation are set to one due to the lack of real values. Thus, the system's WCET is composed of the task's WCETs for a single execution. Other values might influence the optimization decisions but not the result of this evaluation.

The following section discusses the achieved WCET reductions whereas Section 4.4.6.2 gives an insight into required optimization times.

4.4.6.1 WCET Estimations

Figure 4.6 shows the relative WCET$_{est}$s for the benchmark suite *DSPstone*, achieved by our novel optimization presented in Section 4.4.5 as a percentage of the WCET$_{est}$

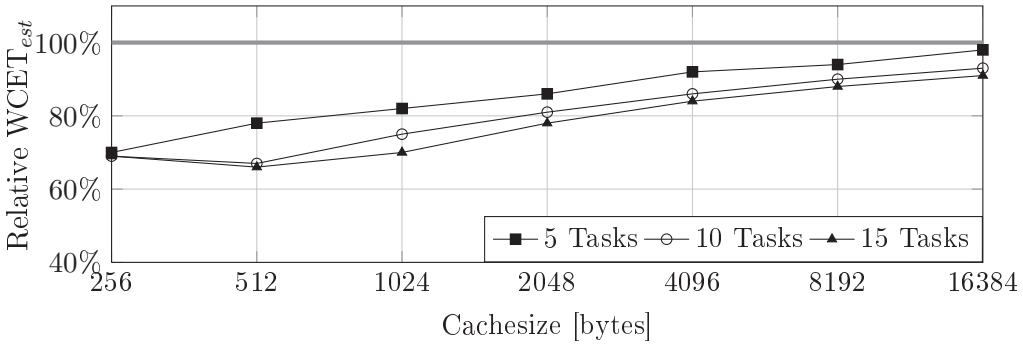


Figure 4.7: Optimized WCET$_{est}$ for MRTC relative to standard approach

achieved by the standard heuristic presented in Section 4.4.4. Thus, the 100% line represents the results achieved by this size-driven partition size selection approach.

The nominal sizes of the task sets range on average from 1.5 kB for 5 tasks up to 5 kB for 15 tasks. Substantial WCET$_{est}$ reductions can only be obtained for smaller caches of up to 1 kB since almost all tasks fit into the cache from 4 kB on. There, WCET$_{est}$ reductions between 4% and 33% can be observed. In general, larger task sets result in higher optimization potential for all cache sizes.

Figure 4.7 depicts the average WCET$_{est}$ for the *MRTC* benchmark suite. The average code size of the generated task sets is comparatively large with 6 kB for 5 tasks, 12 kB for 10 tasks and 19 kB for 15 tasks. Hence, there is more potential to find a better distribution of partition sizes. This can be seen in a nearly linear correlation of the exploited optimization potential and the quotient of task set size and cache size. For 5 tasks in a set, WCET$_{est}$ reductions up to 30% can be gained. 10 tasks per set exhibit a higher optimization potential, so that 7% to 31% decrease of WCET$_{est}$ can be observed. Optimizing the sets of 15 tasks, 9% up to 31% lower WCET$_{est}$s can be achieved.

A similar situation can be observed in Figure 4.8 for the *UTDSP* benchmark suite. The average code sizes for the task sets range from 9 kB to 27 kB. For a 5 task set, this leads to an optimization potential of 25% WCET$_{est}$ reduction for the smallest cache size down to 4% if all tasks completely fit into the 16 kB cache. For a 15 task set, between 17% and 36% lower WCET$_{est}$s can be achieved. For this benchmark suite, the same behavior can be observed: for smaller cache sizes and larger code sizes, our algorithm achieves better results compared to the standard approach.

Using caches larger than 16 kB, our algorithm is not able to achieve better or only marginally better results than the standard method from Section 4.4.4. This comes from the fact that there is almost no optimization potential if all tasks fit into the cache. For realistic applications, the cache would be much smaller than the amount of code. There is also no case where the standard algorithm performs

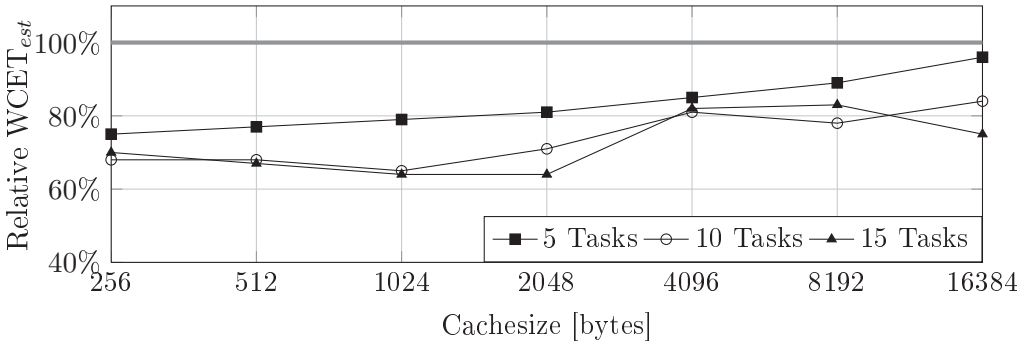


Figure 4.8: Optimized WCET$_{est}$ for UTDSP relative to standard approach

better than our approach since we use ILP models to always obtain the optimal partition size distribution. Since all presented results are improvements compared to established cache partitioning optimizations based on the tasks' sizes, this eveluation eminently demonstrates that our optimization outperforms state-of-the cache partitioning techniques.

4.4.6.2 Optimization Time

To consider compilation and optimization time on the host system, we utilize an Intel Xeon X3220 (2.40 GHz). A complete toolchain iteration is decomposed into the three phases compilation, WCET analysis and optimization. The stage WCET analysis comprises all *aiT* invocations necessary to compute the tasks' WCETs for possible partition sizes and requires most of the overall compilation time. For a task set T and a set of considered partition sizes P, the number of required WCET analyses to set up Equations (4.4) and (4.5) amounts to:

$$\#Analyses_{WCET} = |T| * |P| \tag{4.7}$$

The time required for a combined compilation and optimization phase ranges from less than one second (*fir* from MRTC) to 30 seconds for *adpcm* from UTDSP. Compared to this, the duration for performing static WCET analyses used for construction of an ILP is significantly higher (up to 10 hours).

The work presented in this chapter has been published in [95].

4.5 WCET-driven Multi-Task Program Scratchpad Allocation

Scratchpad memory allocation techniques are widely studied in the context of average-case and worst-case execution time optimization as well as the reduction of

energy consumption [123]. But especially concerning scratchpad allocation of program code for WCET reduction, either only single-task systems can be optimized by an optimal approach [32] or multi-task systems can be handled by a time-consuming iterative approach considering the *worst-case response time of tasks* (*WCRT*) as objective [108].

The WCC compiler already provides an optimal program scratchpad allocation for single tasks [32]. In order to enable the optimization of an entire embedded system, this chapter presents a heuristics-based approach for static program scratchpad allocation which extends the existing technique by multi-task capabilities. For a set of tasks, different heuristics can determine an individual scratchpad memory size per task. For these scratchpad sizes, the existing program scratchpad allocation determines the optimal set of basic blocks to be allocated to the SPM in order to minimize the tasks' WCET of the task.

The remainder of this chapter is organized as follows: Section 4.5.1 motivates the technique of multi-task scratchpad allocation by an example, whereas the following section provides a brief overview of related work. The optimization heuristics employed for selecting the tasks' scratchpad sizes are presented in Section 4.5.3. An evaluation of the effectiveness of the proposed multi-task scratchpad allocation closes this chapter.

4.5.1 Motivating Example

In a multi-task system, caches are shared among all executed tasks except for the case that certain tasks are excluded from caching if, for instance, an extended cache-aware memory content selection as presented in Section 3.4 is applied. Techniques such as the cache partitioning (cf. Section 4.4) can make multi-task systems predictable but also limit the available cache space per task.

In order to release pressure from the cache, fast scratchpad memories can be employed to store frequently used code or data objects. Since these memories are rather small, a skillful content selection is crucial in order to achieve a high performance. For average-case execution time optimizations, this problem can be reduced to solving the Knapsack problem (refer to Appendix A.1.4 for a definition). The size of the objects to allocate represents the weights whereas the gain is represented by the improvement w. r. t. the considered objective – here the saved runtime in processor cycles.

Optimizing the WCET of a program is a more complex task since possible WCEP switches can have direct influence on the gain of a certain memory object. Iterative allocation approaches cannot guarantee an optimal solution w. r. t. the considered objective. Falk et al. therefore developed an ILP-based approach which models the influence of allocation decisions on the WCEP and the resulting WCET similar to the modeling presented in Section 3.3.3.2, page 41. This approach is best suited for the optimization of a single task running on a system but is not aware of partitioning a scratchpad for a multi-task usage.

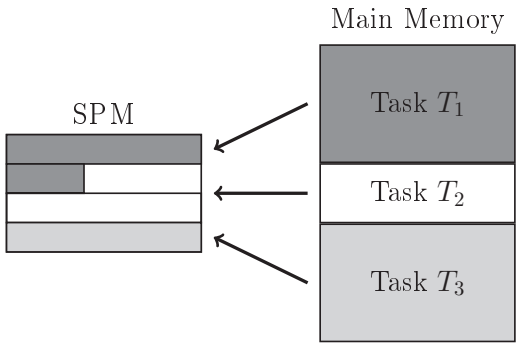


Figure 4.9: Scratchpad memory allocation problem

In a multi-task scenario, the question is not only which objects of a task should be allocated to the scratchpad but also how much memory area is reserved for a certain task. Such a partitioning problem is similar to the partition size selection in the context of cache partitioning (cf. Section 4.4.5). But unlike caches, scratchpads can be divided in a finer-grained manner. As depicted in Figure 4.9, the scratchpad can be divided at almost arbitrary boundaries (usually word aligned). The selected scratchpad partition size can have a direct impact on the allocation decision of conventional scratchpad allocation techniques.

For that reason, Section 4.5.3 presents techniques to select partition sizes for a given set of tasks. Afterwards, a conventional scratchpad allocation approach can be separately applied to each single task for a given partition size.

4.5.2 Related Work

One of the first works exploring the effect of program scratchpads on the WCET prediction is presented in [122]. Wehmeyer et al. employed an ILP model which minimizes the overall energy consumption of a program. Afterwards, the trend in terms of runtime and WCET is evaluated. Unfortunately, the WCET is not taken into account as optimization metric, and the approach is limited to single task optimizations.

Verma et al. propose partitioned program scratchpads for the optimization of multi-task embedded systems [120]. Either a static or a dynamic partitioning or a combination of both is employed to divide the scratchpad. An ILP is employed to select the optimal partition size w. r. t. the overall energy consumption of a system. Nevertheless, the ILP is not able to take the WCET and switching WCEPs into account.

In [107], Suhendra et al. reduce the WCET of programs by allocating frequently accessed data objects to faster memories. A program's possible execution paths are modeled in an ILP in order to consider its WCET. Since only the intra-function control flow is modeled, a time-consuming branch-and-bound approach or a sub-

optimal heuristic is employed to optimize along the WCEP. Furthermore, allocation of program code is left out as well as the optimization of multi-task systems.

Decreasing the WCET is also the objective of Falk's program scratchpad allocation [32] which is exploited by our multi-task scratchpad allocation presented in this chapter. Similar to [107], a program's execution paths and their influence on the resulting WCET is modeled. But in contrast to [107], the overall control flow is modeled in order to optimize along the WCET. Thereby, optimization requires no time-consuming iterative techniques, and only a single ILP has to be solved to determine the optimal combination of basic blocks for the scratchpad. As mentioned above, only single tasks can be optimized.

Another work of Suhendra [108] considers the program scratchpad allocation for multi-task systems. A time-consuming iterative approach is required to consider the *worst-case response time* ($WCRT$) of the tasks as objective. In contrast to the work presented in this chapter, a dynamic allocation is performed which means that reloading of scratchpad content is performed at appropriate program points. Such a dynamic behavior complicates a static timing analysis and often forbids such a scratchpad allocation technique for hard real-time systems.

4.5.3 Multi-Task Program Scratchpad Allocation

The fundamental idea of our multi-task scratchpad allocation is to exploit the sophisticated WCET-driven static program scratchpad allocation for single tasks [32]. The single-task version achieves high WCET reductions since its ILP-based approach always optimizes along a program's WCEP. A program to optimize as well as the available scratchpad size is required as input.

The problem of selecting the optimal SPM partition size for each task in a set leading to the maximum reduction of the WCET is similar to the cache partition size selection presented in Section 4.4. In a similar fashion, a single-task scratchpad allocation could be performed for each task and each considered partition size. Based on the resulting WCETs, an ILP could select the optimal SPM partition sizes w. r. t. the overall WCET of a system. Since a single-task scratchpad allocation performs two separate WCET analyses during optimization, the drawback of such an ILP-based approach would be a doubling of the anyway high optimization times (cf. Section 4.4.6.2).

Therefore a multi-task extension was developed which employs heuristics determining the assigned program scratchpad size for each task based on static features in order to decrease the WCET of the entire system. For each of the tasks, a single-task scratchpad allocation is invoked with the determined scratchpad size as parameter. In the following, the different heuristics based on characteristics of the programs to optimize, namely the WCETs, the code sizes and a combination of both, are presented.

WCET-based Heuristic

The basic idea of the WCET-based heuristic is that tasks which highly contribute to the system's overall WCET obtain larger parts of the scratchpad than tasks with lower WCETs. Such a strategy expects that the size of the frequently executed program parts correspond with the WCET of a program. Programs with a high WCET hence should have a high code size on the WCEP and probably benefit from a large assigned scratchpad partition.

The available scratchpad memory is distributed amongst the tasks depending on their share in the system's overall WCET. For a set T of tasks, first of all, the WCET $w(T_i)$ for each task T_i has to be determined by employing a static timing analyzer. According to the task's execution frequency f_i in a hyperperiod H (cf. Definition 4.1, Section 4.4.4) and to the scratchpad memory size S_{SPM}, T_i's partition size computes as follows:

$$p(T_i) = \frac{f_i * w(T_i)}{\sum_{j=1}^{|T|} f_j * w(T_j)} * S_{SPM} \tag{4.8}$$

A tasks T_i's overall WCET in a hyperperiod is the products of its WCET_{est} and its execution frequency within H. Thus, the ratio of this resulting WCET to the system's overall WCET correlates with T_i's share in the SPM.

Code Size based Heuristic

Code size based partition size selection assumes that tasks with larger code sizes potentially have more program blocks which contribute to the WCET of the task. Of course, this strategy only succeeds if there are only few dead code blocks and all parts of the program contribute to the WCET as equal as possible.

Depending on a task's share in the systems overall code size, the scratchpad memory is divided into partitions. For a given set T of tasks, task T_i's code size is denoted $s(T_i)$. Regarding a scratchpad memory size S_{SPM}, T_i's partition size $p(T_i)$ computes as follows:

$$p(T_i) = \frac{f_i * s(T_i)}{\sum_{j=1}^{|T|} f_j * s(T_j)} * S_{SPM} \tag{4.9}$$

Hybrid Heuristic

Both the WCET- and the code size based heuristic have in common that they rely on assumptions on the program structure and the size of the program parts residing on the WCEP. If these assumptions are incorrect, the heuristic cannot achieve good results. Large programs, for instance, can have only few hot spots such that all parts contribute to the WCET to the same extent. Such programs cannot efficiently be optimized by the WCET-based heuristic. Conversely, the heuristic based on the tasks' code size possibly cannot efficiently optimize small programs which highly contribute to the system's WCET.

Table 4.1: Multi-task benchmark sets

	Benchmark	Code size [bytes]	$WCET_{est}$ [cycles]
*Set*1	g721_encode	4,852	2,278,604
	edge_detect	770	39,072,467
	latnrm_32x64	418	311,380
*Set*2	crc	518	252,160
	g721_marcuslee_decoder	144	231,114
	h264dec_ldecode_block	13,994	325,171
*Set*3	trellis	3,086	1,187,540
	fft1	1,936	91,982
	gsm_decode	6,532	17,371,046
	crc	518	252,160
*Set*4	h264dec_ldecode_block	13,994	325,171
	g721_marcuslee_decoder	144	231,114
	crc	518	252,160
	histogram	576	387,459
	latnrm_32x64	418	311,380

In order to cover such cases, a hybrid heuristic is employed which tries to take the influence of a task's code size as well as its WCET into account. For a set of tasks comprising $|T|$ tasks, the task's code size $s(T_i)$, their WCET $w(T_i)$, their execution frequency f_i within a hyperperiod and the scratchpad memory size S_{SPM} is given. Then, T_i's partition size $p(T_i)$ computes as follows:

$$p(T_i) = 0.5 * \left(\frac{f_i * s(T_i)}{\sum_{j=1}^{|T|} f_j * s(T_j)} + \frac{f_i * w(T_i)}{\sum_{j=1}^{|T|} f_i * w(T_j)} \right) * S_{SPM} \tag{4.10}$$

Independent of Equations (4.8) – (4.10), all presented heuristics have in common that the partition size per task can only grow up to the task's code size. This avoids wasting of scratchpad memory since the remaining space is distributed among the remaining tasks.

4.5.4 Evaluation

This section evaluates the effectiveness of the different heuristics presented in the preceding section. All evaluations are performed for the TriCore TC1796 with the WCC at optimization level *O3* (refer to Section 2.4.7 for details). As stated in Section 2.5, content allocated to the scratchpad can be accesses with on cycle latency whereas the content of the flash can be accessed with six cycles latency for the first access and two cycles for directly following accesses to the same memory line. The heuristics for multi-task scratchpad allocation are applied to different sets of benchmarks stemming from various benchmark suites. Table 4.1 lists the employed

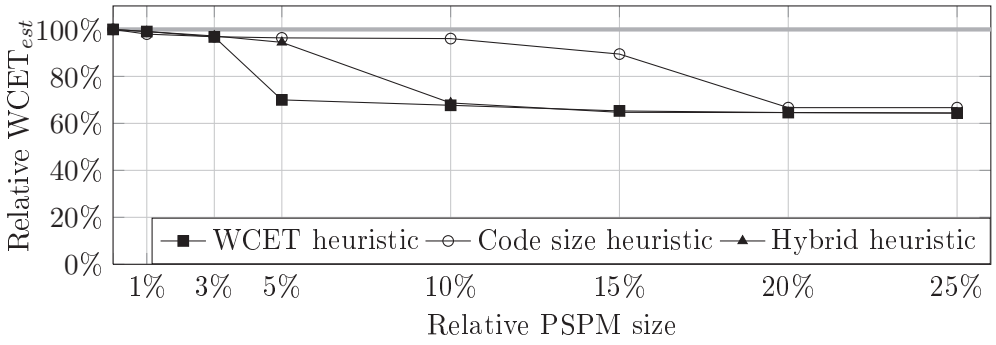


Figure 4.10: Optimized WCET$_{est}$ for multi-task PSPM allocation applied to *Set*1

four benchmark sets comprising three up to five benchmarks implementing typical tasks in multimedia embedded systems. Cellular phone codecs are combined with several audio and video codecs with widely spread code sizes ranging from 144 bytes (*g721_ marcuslee_ decoder*) up to 13 kB (*h264dec_ ldecode_ block* kernel). Moreover, the WCETs of the benchmarks differ by a factor of up to 425 (*fft1* vs. *edge_ detect*).

All evaluations are performed for instruction scratchpad sizes from 0% up to 25% of the overall program size of the task set. Hence, the tested range of SPM sizes features a reasonable ratio of scratchpad memory to main memory and can be found in most of the embedded systems equipped with SPM. As done in Section 4.4.6, the tasks' execution frequencies f_i are set to one since synthetic benchmark sets are employed for which no real values exist.

The following Section 4.5.4.1 presents the evaluation of the WCET estimates for all benchmark sets whereas Section 4.5.4.2 gives an overview of time required for compilation and optimization.

4.5.4.1 WCET Estimations

Figure 4.10 depicts the results if the heuristics are applied to benchmark *Set*1. The x-axis represents the evaluated program scratchpad memory sizes (PSPM) relative to the overall program size of the entire benchmark set. The achieved relative WCET$_{est}$ of the benchmark set is shown on the y-axis as percentage of the WCET$_{est}$ in a system without SPM. Thus, the 100% line represents the sum over the WCET$_{est}$s of the benchmark in a set as depicted in Table 4.1. All proposed heuristics are able to reduce the WCET$_{est}$ of the considered benchmark set by up to 35.6% for SPM sizes from 20% on, whereas the code size heuristic achieves slightly lower reductions of 33.3%. For smaller scratchpad sizes, the code size heuristic performs considerably worse compared to the others. On average, the WCET-based heuristic performs best since it is able to reduce the WCET$_{est}$ by 30.0% even for small SPMs of 5%.

The results indicate that the size of the program code on the critical path (the WCEP) highly correlates to the WCET$_{est}$ of the tasks and does not depend on

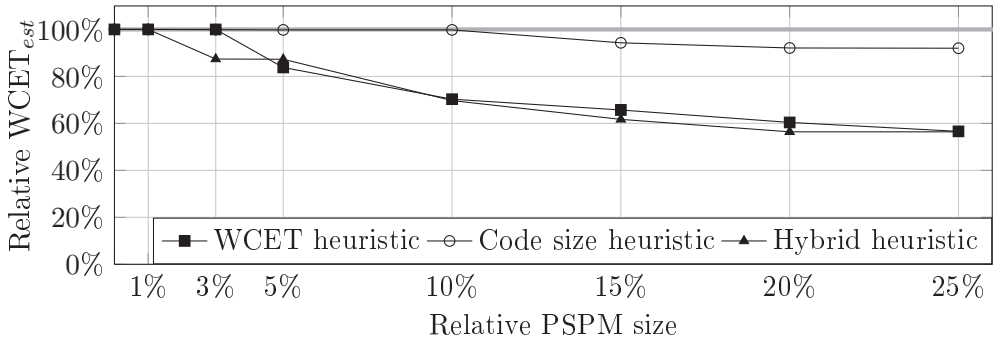


Figure 4.11: Optimized WCET$_{est}$ for multi-task PSPM allocation applied to $Set2$

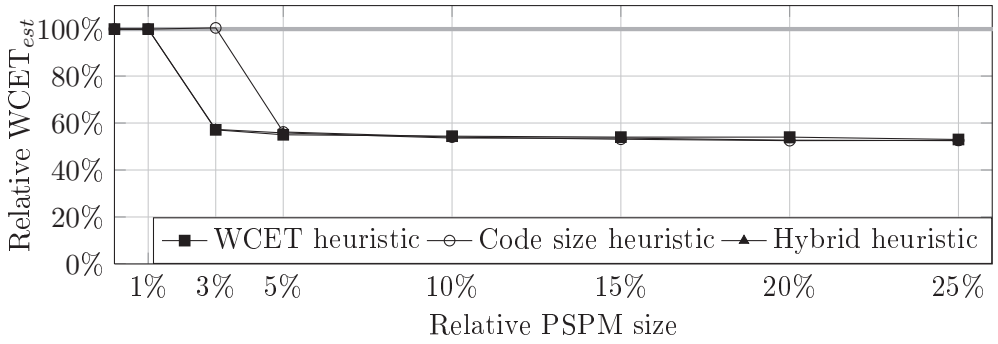


Figure 4.12: Optimized WCET$_{est}$ for multi-task PSPM allocation applied to $Set3$

their code sizes. Otherwise, the code size based and hybrid heuristic would perform better.

The results for applying the three heuristics to benchmark $Set2$ are shown in Figure 4.11. Now, the maximally achievable WCET$_{est}$ reductions are widely spread. The hybrid heuristic as well as the WCET-based heuristic achieve the highest WCET$_{est}$ reductions of up to 43.4% for 25% SPM size. The code size based heuristics only reduces the WCET$_{est}$ by up to 8.0%. For smaller SPM sizes, the same behavior can be observed: the code size based heuristic is outperformed by the WCET-based heuristic which achieves similar results compared to the hybrid heuristic. Only for 3% SPM size, the hybrid heuristic achieves WCET$_{est}$ reduction of 12.6% whereas the other heuristics cannot reduce the WCET$_{est}$.

Comparing the achieved WCET$_{est}$ reductions shows that the results of the hybrid heuristic is a linear combination of the code size and WCET-based heuristics. In contrast to $Set1$, the size of the program code of $Set2$ which resides on the WCEP does not only depend on the WCET$_{est}$ of the tasks or their code sizes. Instead, both factors influence the critical program size which makes the hybrid heuristic powerful in selecting the SPM partition size for each task.

In Figure 4.12, the results for the proposed heuristics and benchmark $Set3$ are shown. All heuristics are able to achieve similar WCET$_{est}$ reductions: The hybrid

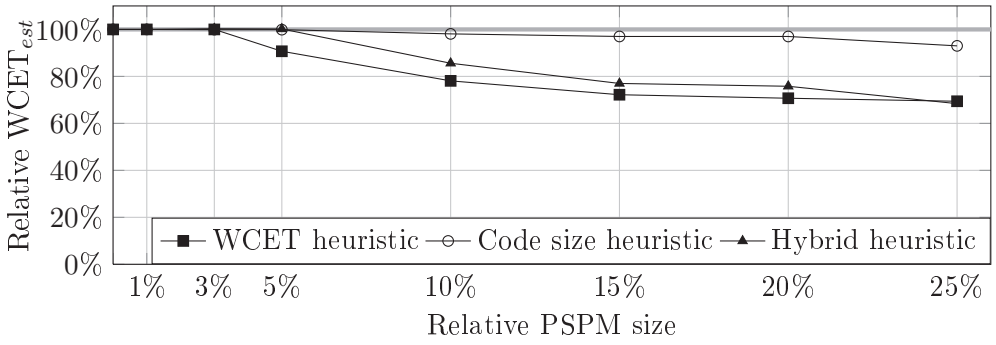


Figure 4.13: Optimized WCET$_{est}$ for multi-task PSPM allocation applied to *Set*4

heuristic is able to achieve WCET$_{est}$ reductions of up to 47.5% whereas the WCET-based and the code size heuristic reduce the WCET$_{est}$ by up to 47.0% and 47.4%, respectively. Substantial WCET$_{est}$ reductions are achieved by all heuristics from 5% SPM size on. At this point, the WCET-based heuristic marginally outperforms the others by 1%. For *Set*3, no clear advantage of one heuristic can be figured out – only the code size heuristic cannot be recommended since it performs worse for 3% SPM size.

Finally, Figure 4.13 depicts the results if the proposed heuristics are applied to *Set*4. The WCET-based and the hybrid heuristics perform best since the WCET$_{est}$ can be reduced by up to 31.5% whereas the code size based heuristic achieves only WCET$_{est}$ reductions by up to 7%. In contrast to the other benchmark sets, substantial WCET$_{est}$ reductions are achieved from 15% SPM size on. Again, the hybrid heuristic is a linear combination of the WCET-based and the code size based heuristic – but outperformed by the WCET-based heuristic.

The results for the four benchmark sets clearly indicate that the WCET heuristic performs best. The hybrid heuristic performs worse – except for benchmark *Set*2 – but considerably outperforms the code size based heuristic.

4.5.4.2 Optimization Time

In order to consider the optimization time required for multi-task scratchpad allocation, an Intel Xeon E5650 (2.67 GHz) was utilized. As observed for other WCET optimizations presented so far, most of the time necessary for our novel WCET-driven multi-task scratchpad allocation was consumed by WCET analyses using *aiT*. As for the single task SPM allocation presented in [32], the WCET of each task has to be analyzed once residing in main memory and once in SPM. Thus, the maximal number of WCET analyses for a set T of tasks amounts to:

$$n = 2 * |T| \tag{4.11}$$

The computational power for determining the share of each task in scratchpad memory (cf. Equations (4.8) – (4.10)) is negligible compared the time spent for

estimating the WCET and applying the single task SPM allocation. Table 4.2 depicts the optimization times for each task without distinguishing the different heuristics for SPM size selection.

Depending on the complexity of the task set, the required optimization time ranges from 9 minutes for $Set2$ up to 57 minutes for $Set3$.

Table 4.2: Optimization times for multi-task SPM allocation

Benchmark Set	Optimization Time [s]
$Set1$	537
$Set2$	454
$Set3$	3,397
$Set4$	532

4.6 Memory Architecture aware Compilation

It is common practice that WCET-driven optimizations are developed stand-alone and are applied separately at the very end of the optimization process. Only a single WCET optimization is usually performed after applying various standard compiler optimizations. This strategy avoids unintended interactions of, for instance, memory-based optimization techniques but also wastes optimization potential: synergetic effects by an intelligent combination of existing WCET optimizations cannot be exploited.

To the best of the author's knowledge, this chapter for the first time explores the possible optimization potential of combined WCET-driven optimization techniques and gives recommendations for the order in which memory-based optimizations should be applied. The optimizations presented in Sections 3.4, 4.4 and 4.5, namely *cache-aware memory content selection* (**MCS**), *software based cache partitioning* (**CP**) and *multi-task program scratchpad allocation* (**MPSPM**), are employed in this chapter. The optimizations are examined for their impacts on the memory layout of a program, and a convenient application order is derived which exploits modifications of preceding optimizations. Necessary modifications are separately discussed for each optimization.

The remainder of this chapter is organized as follows: The following Section 4.6.1 motivates the need of dedicated optimization sequence considerations by an example. Section 4.6.2 provides an overview of research which is related to the work presented in this chapter. Section 4.6.3 presents reflections about how to derive a convenient order of memory-based optimizations whereas Section 4.6.4 describes the required modifications of the individual optimizations in such a chain. Finally, an evaluation concludes this chapter in Section 4.6.5.

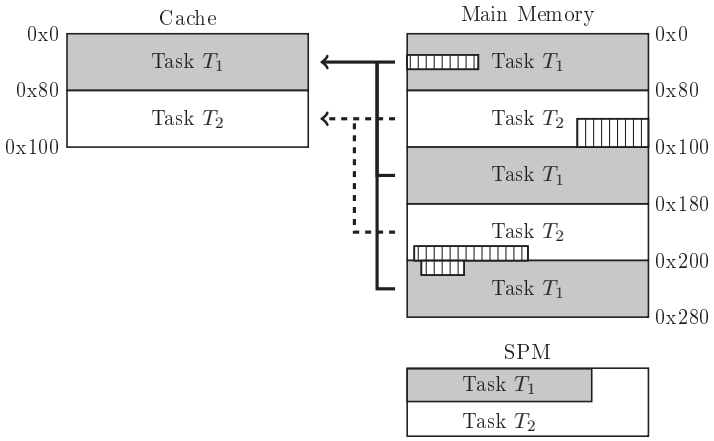


Figure 4.14: Memory layout interference

4.6.1 Motivating Example

Besides the optimizations presented in this thesis, various publications [119, 123, 91, 31] have shown that the memory subsystem is often a performance-limiting factor and entails an immense optimization potential. Such techniques have in common that they improve the worst-case performance of a system by optimizing its memory access patterns. This can be achieved by allocating program or data objects to faster memories, by solving cache conflicts or prefetching of memory objects which will be used in the near future.

Applying a number of memory-based optimizations in an arbitrary order can have adverse effects if an optimization has disadvantageous influence on the memory layout arranged by a preceding optimization. Assumed that software-based cache partitioning (cf. Section 4.4) is applied to a multi-task set, a memory layout as depicted in Figure 4.5, p. 81, can be the result. If multi-task program scratchpad allocation is applied afterwards, the allocation of basic blocks to a scratchpad memory creates gaps in the address space represented by lined boxes in Figure 4.14.

Such parts of tasks T_1 and T_2 which are allocated to the fast SPM cannot easily be filled with the remaining code due to the fixed binding of task partitions to certain memory addresses. Thus, cache area where the moved code blocks have been mapped before may remain unused (ruled boxes). In view of the tasks' decreased code sizes residing in main memory, the cache partitioning would probably make different partition size selection decisions since otherwise optimization potential is wasted. Hence, it would be more sensible to perform a scratchpad memory allocation before applying a cache partitioning technique. The cache partitioning, however, has to regard preceding allocation decisions and must not move basic blocks located in the SPM.

4.6.2 Related Work

Optimization sequence determination is also termed as *phase-ordering problem* by Touati et al. [114]. The authors formalize the problem of finding an improved order of optimization phases achieving the optimal performance. The presented approach does neither analyze the interference of memory-based optimization nor considers the WCET as metric.

In [101], Queva presents a phase manager which helps to solve the phase-ordering problem. Based on data-flow analyses, metrics are calculated which are used to evaluate where a certain transformation can be applied. Unfortunately, the presented framework is not able to consider memory-based optimization techniques since most of them have no impact on the data-flow of a program.

Ishizaki et al. introduce a Java Just-In-Time compiler and ranked existing optimizations w. r. t. the performance improvement for three target platforms [58]. Compared to the work presented in this chapter, their approach considers optimizations as black boxes and does not analyze the mechanics behind which can lead to interference between optimizations. The order of optimizations is not changed at all.

A work considering the influence of compiler optimization sequences is considered by Almagor et al. in [3]. Besides a simple heuristic, an evolutionary approach is employed to find improved optimization sequences which outperform handcrafted optimization levels of standard compilers. Nevertheless, no insight into interference of optimizations is provided, and the WCET is not considered as metric.

Lokuciejewski et al. extended the ideas presented in [3] in order to find improved optimization sequences for multi-objective optimizations [78]. Although WCET reductions are considered besides ACET and code size savings, their approach only utilizes well-known standard ACET optimizations which are not tailored to reduce the WCET. The interference of particular optimizations is only implicitly considered by their influence on the considered objectives.

Up to now, only Srikant et al. examine possible interactions of compiler optimizations and provide at least partial knowledge of these mechanisms [104]. Unlike the work presented in the following, no work considers the application order of WCET-driven optimizations and/or provides modifications for existing optimizations in order to improve their interactions.

4.6.3 Optimization Sequence Determination

In order to make statements about promising optimization sequences, the influence of the considered optimizations has to be evaluated. All optimizations have in common that they modify the memory layout of tasks in a certain way: the scratchpad memory allocation moves basic blocks to a fast SPM in order to speed up their execution. The memory content selection and the cache partitioning, in contrast, optimize the cache access pattern of tasks. Thus, it would be harmful if the SPM

allocation is applied after any cache-based optimization and thereby disturbs the optimized cache access patterns.

Concerning the application order of the two cache-based optimizations, a mutual dependence exists. On the one hand, the cache partitioning algorithm would possibly select different partition sizes if the memory content selection omits functions from caching. But on the other hand, the memory content selection algorithm is sensitive for the available cache size and thus for the partition size determined by the cache partitioning. Hence the following optimization sequence is proposed:

1. **Scratchpad memory allocation**

   Applied as first WCET optimization, the SPM allocation is able to move such basic blocks to the fast SPM which contribute most to the WCET. This also has two positive side-effects: firstly, the cache pressure is reduced since less code blocks are allocated to the cached memory and can compete for the same cache lines. Secondly, the SPM allocation does not interfere with the succeeding cache-based optimizations.

2. **Software based cache partitioning**

   As second optimization, cache partitioning is applied which is able to process and optimize a multi-task set. To set up Equations (4.3) to (4.6), p. 84, the WCET for each task and each considered cache size has to be determined. Thus, it seems to be promising to apply the memory content selection for each task and each partition size before estimating the WCET.

3. **Memory content selection**

   Since the optimization is not aware of partitioned caches, the original version cannot be employed. On this account, the memory content selection is integrated into the cache partitioning optimization which is described in the following.

4.6.4 Modification of Optimizations

This section describes the modifications required for a fine-tuned optimization sequence comprising the above mentioned WCET-driven memory-based optimizations. Since the scratchpad memory allocation is applied as the very first one, no modification of the original implementation is necessary.

The idea for a tight integration of the WCET-aware memory content selection into the software based cache partitioning is sketched in Figure 4.15. The cache partitioning optimization starts to iterate over all *task entries* of the *task set* under optimization and has to determine the WCET for each *task entry* and each considered partition size (cf. Section 4.4.5). For this purpose, a modified memory content selection is employed which tries to determine the optimal set of cached functions for the actual considered *task entry* and partition size.

Therefore, memory content selection is extended to be aware of partitioned caches. For this purpose, the mechanism to partition a cache in software was integrated into the optimization. If an allocation decision is made, the set of functions to be placed in the cache memory is partitioned according to the current partition size. Afterwards, the resulting WCET of the actual task can be evaluated for the current set of cached/non-cached functions under the current cache partition size.

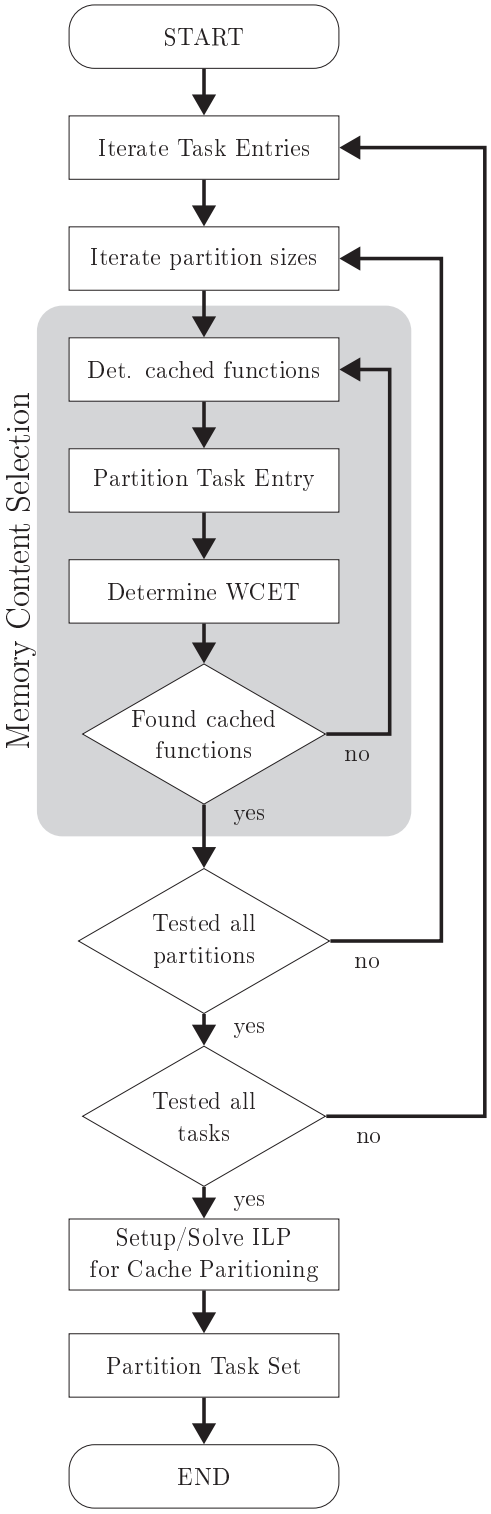


Figure 4.15: Schematic workflow of combined cache partitioning and memory content selection

If a promising set of cache functions has been found, the memory content selection terminates.

Then, the cache partitioning optimization continues with determining the WCET for the remaining partition sizes and *task entries* by invoking the memory content selection repetitively. If the required WCET data for each task and each partition size is collected, the ILP is set up and solved. Finally, the task set is partitioned according to the solutions gathered from the ILP.

At the same time, the flow diagram in Figure 4.15 also reveals the drawback of such a combined optimization approach: the necessary – but time consuming – WCET analyses for each task and each partition size is performed within a threefold nested loop. Thereby, the number of WCET analyses is multiplied which increases the optimization time. An evaluation of the achieved WCET reductions as well as the required optimization time for combined WCET optimizations is presented in the following section.

4.6.5 Evaluation

This section evaluates the effectiveness of the presented memory architecture aware compilation based on the optimizations *multi-task scratchpad memory allocation*, *cache partitioning* and *memory content selection*. The environment is similar to the one presented in Section 4.5.4: WCC is employed to generate code for the Infineon TriCore processor TC1796 at optimization level *O3*. The same set of benchmarks as depicted in Table 4.1, p. 93, is utilized. The employed SPM size is varied from 0% up to 25% of the task sets' overall program size. This is done in order to gather results comparable to the stand-alone multi-task scratchpad allocation presented in Section 4.5. The explored cache size ranges from 0% up to 25% of the task sets' overall program size as well. This supports a comparison whether one of the memories is more useful for improving the WCET of multi-task sets or if a combination of both is worthwhile.

A separate evaluation of the employed WCET optimizations was presented in Section 3.4.4, p. 65, for the cache-aware memory content selection, in Section 4.4.6, p. 85 for the software based cache partitioning and in Section 4.5.4, p. 93, for the multi-task program scratchpad allocation. In the following, possible and useful combinations are considered in order to evaluate the effect of individual optimizations. Therefore, Section 4.6.5.1 evaluates the influence of combined cache partitioning and memory content selection on the WCET of multi-task sets whereas Section 4.6.5.2 tests the combination multi-task scratchpad memory allocation and cache partitioning. Section 4.6.5.3 combines both latter with memory content selection. Finally, Section 4.6.5.4 highlights the optimization time necessary for combined WCET optimizations.

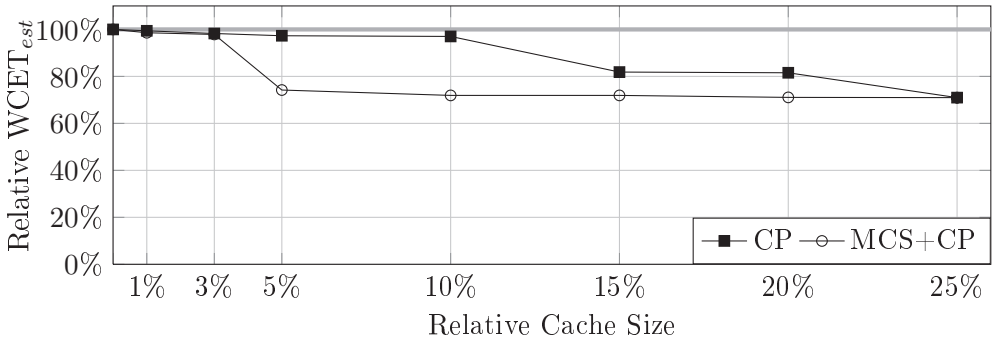


Figure 4.16: Optimized WCET for Cache-aware Memory Content Selection and Cache Partitioning applied to *Set*1

4.6.5.1 WCET Estimations for Cache Partitioning combined with Memory Content Selection

Since *MPSPM* allocation is applied as the very first optimization, no interference with the following optimizations is expected. Thus the application of *MCS* coupled with *CP* is examined in this section. Figures 4.16 and 4.17 compare a combined application of both optimizations with applying only cache partitioning to *Set*1 and *Set*3, respectively. Applying the optimizations to *Set*2 and *Set*4 yields similar results which are not explicitly discussed in the following. Full results are given in Figures A.1 and A.2 in the Appendix.

The x-axis in Figure 4.16 shows the employed cache size as percentage the overall program size of the task set whereas the y-axis depicts the resulting WCET$_{est}$ compared to the unoptimized version. The 100% line represents this unoptimized version of a system running without any cache. Applying cache partitioning achieves WCET$_{est}$ reductions by up to 29.0% for a cache size of 25% of the overall code size. Combining *MCS* with *CP* achieves the same maximum WCET$_{est}$ reduction. For cache sizes below 5%, only marginal WCET$_{est}$ reductions can be achieved for both combinations of optimizations. But for cache sizes from 5% on, additionally applying *MCS* is worthwhile since it outperforms traditional *CP* by up to 25.0% for 10% cache size. Only if the cache becomes large enough to store all code blocks residing on the the WCEP, traditional *CP* catches up to the combination of *MCS* and *CP*.

Figure 4.17 depicts the result if *CP* and *MCS* are applied to *Set*3. Here, both *CP* and *CP* combined with *MCS* are able to reduce the WCET$_{est}$ by up to 44.7%. Since for cache sizes from 10% on, almost all code residing on the critical path fits into the cache, additionally applying *MCS* does not yield lower WCET$_{est}$s. Only for 5% cache size, *MCS* can decrease the WCET$_{est}$ by additional 6.3% compared to applying only *CP*.

Whether a combined application of cache partitioning and memory content selection is worthwhile or cache partitioning should be applied solely depends on the code size residing on the critical path and cannot be answered for the general case.

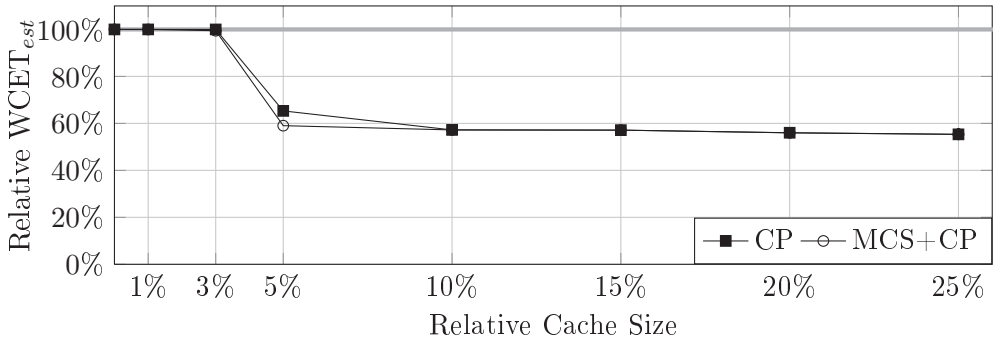


Figure 4.17: Optimized WCET$_{est}$ for Cache-aware Memory Content Selection and Cache Partitioning applied to *Set*3

Table 4.3: Optimized WCET$_{est}$ for MPSPM allocation and CP applied to *Set*1

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	99.4%	98.3%	97.3%	97.0%	85.2%	85.2%	71.2%
1%	99.1%	98.8%	97.3%	96.7%	96.2%	85.2%	82.2%	71.2%
3%	96.8%	96.4%	94.9%	94.0%	93.8%	83.2%	80.6%	70.8%
5%	70.0%	69.6%	68.1%	67.2%	67.0%	67.0%	66.8%	66.8%
10%	67.7%	67.3%	66.1%	64.9%	64.7%	64.7%	64.5%	64.5%
15%	65.3%	65.3%	65.3%	64.7%	64.6%	64.5%	64.4%	64.3%
20%	64.6%	64.6%	64.6%	64.5%	64.4%	64.4%	64.3%	64.2%
25%	64.4%	64.4%	64.4%	64.4%	64.3%	64.2%	64.1%	64.1%

4.6.5.2 WCET Estimations for Multi-Task SPM Allocation combined with Cache Partitioning

The previous section evaluated the influence of combining cache partitioning with memory content selection. This section evaluates the impact if cache partitioning is combined with multi-task scratchpad memory allocation. Therefore, the Cartesian product of cache and SPM sizes from 0% up to 25% are evaluated for *Set*1 and *Set*2. The results are presented in Table 4.3 and 4.4, respectively.

In Sections 4.5 and 4.4.6, it was already demonstrated that applying *CP* or only *MPSPM*, to the benchmark sets achieves WCET$_{est}$ reductions of up to 47.3% and 44.7% for *MPSPM* allocation and cache partitioning, respectively. This section should evaluate if a combined application of both optimizations entails even higher optimization potential. As in preceding sections, the SPM size and the cache size are varied from 0% to 25% of the overall code size of the considered task set. A full evaluation of the different heuristics for *MPSPM* allocation is omitted at this point. Instead, the most powerful WCET-based heuristic is employed.

Table 4.3 presents the achieved WCET$_{est}$s for *Set*1. The utilized cache size for each cell is depicted at the column header whereas the SPM size for each row is de-

Table 4.4: Optimized $WCET_{est}$ for MPSPM allocation and CP applied to $Set2$

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	100.0%	97.2%	91.9%	80.2%	77.5%	76.8%	69.1%
1%	100.0%	100.0%	97.2%	91.9%	80.2%	77.5%	76.8%	69.1%
3%	100.0%	83.5%	75.8%	70.3%	64.2%	60.5%	59.7%	56.3%
5%	83.8%	66.5%	63.9%	61.2%	55.1%	53.2%	52.4%	51.9%
10%	70.3%	65.7%	63.0%	58.9%	52.5%	52.4%	51.8%	51.8%
15%	65.7%	62.4%	53.6%	53.0%	52.5%	52.3%	51.6%	51.6%
20%	60.4%	60.1%	53.6%	52.8%	52.5%	52.3%	51.7%	51.7%
25%	56.6%	56.6%	53.6%	52.8%	52.5%	52.3%	51.7%	51.7%

picted in the first column. The 0% column respective 0% row represent the results if the cache partitioning and multi-task scratchpad allocation optimizations, presented in Section 4.4 and 4.5, are applied separately. With growing SPM/cache sizes, the $WCET_{est}$ of the task set is decreased. The highest $WCET_{est}$ decrease of 35.9% is achieved for 25% SPM size and cache sizes from 20% on. *MPSPM* allocation is better suited than *CP* to optimize this task set since even small SPM sizes of 5% are able to achieve reductions of 30% w. r. t. $WCET_{est}$. In this case, a combination with *CP* does not pay off since only marginal $WCET_{est}$ improvements of 0.4% – 3.2% can be observed for larger cache sizes if combined with *MPSPM* allocation. This is caused by the program structure of the tasks which exhibits few hot spots which can be simply allocated to an SPM but otherwise would cause cache thrashing by mutual eviction from the cache.

The benefits achieved by applying *CP* and *MPSPM* allocation to $Set2$ are listed in Table 4.4. The $WCET_{est}$ of the task set is decreased with growing SPM and cache sizes whereas a maximum $WCET_{est}$ reduction of 48.4% can be achieved for 15% SPM and 20% as well as 25% cache size. Although no interference between the *MPSPM* allocation and *CP* was expected, the results for 20% SPM and 20% cache size proves the opposite: The $WCET_{est}$ slightly worsens by increasing the SPM from 15% to 20% of the overall task set size at unchanged 20% cache size. An altered memory layout caused by a different allocation of code blocks to the larger SPM can lead to a completely different cache behavior if new cache conflicts arise. In view of the marginal $WCET_{est}$ increase of 0.1%, this interference is negligible.

In contrast to $Set1$, the $WCET_{est}$ of $Set2$ is not induced by small hot spots. Large parts of the code contribute to the $WCET_{est}$ leading to a continuous $WCET_{est}$ decrease if the SPM and cache size are increased. Here, a combination of *CP* with *MPSPM* is worthwhile since the results outperform a separate application of the individual optimizations by 4.9% (compared to *MPSPM*) and by 17.4% (compared to *CP*). The best compromise seems to be a cache size and SPM size of 10% resulting in a $WCET_{est}$ reduction of 47.5%.

Table 4.5: Optimized $WCET_{est}$ for MPSPM allocation, CP and MCS applied to $Set1$

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	98.6%	97.9%	74.2%	71.9%	71.9%	71.1%	71.1%
		(99.4%)	(98.3%)	(97.3%)	(97.0%)	(85.2%)	(85.2%)	
1%	99.1%	97.8%	97.1%	75.1%	72.8%	72.8%	71.2%	71.2%
		(98.8%)	(97.3%)	(96.7%)	(96.2%)	(85.2%)	(82.2%)	
3%	96.8%	95.5%	94.8%	74.0%	71.7%	71.7%	70.9%	70.9%
		(96.4%)		(94.0%)	(93.8%)	(83.2%)	(80.6%)	
5%	70.0%	68.6%	67.9%	67.7%	67.1%	67.1%	66.8%	66.8%
		(69.6%)						
10%	67.7%	66.3%	65.6%	65.4%	64.7%	64.7%	64.5%	64.5%
		(67.3%)	(66.1%)					
15%	65.3%	65.3%	65.1%	64.6%	64.4%	64.4%	64.3%	64.3%
20%	64.6%	64.6%	64.5%	64.4%	64.3%	64.3%	64.2%	64.2%
25%	64.4%	64.4%	64.3%	64.2%	64.1%	64.1%	64.0%	64.0%

The results for $Set3$ and $Set4$ exhibit similar results as $Set1$ and $Set3$. A detailed discussion is omitted at this point; the interested reader is referred to Appendix A.1 and A.2 for detailed results.

4.6.5.3 WCET Estimations for Multi-Task SPM Allocation, Cache Partitioning and Memory Content Selection

The previous section evaluated the impact of multi-task program scratchpad allocation combined with cache partitioning. The evaluation presented in Section 4.6.5.1 pointed out that the effectiveness of *CP* can be improved by combining it with memory content selection. Hence, this section evaluates the effectiveness if *MPSPM* allocation, *CP* and *MCS* are combined as described in Section 4.6.3.

Table 4.5 depicts the results if the three optimizations are applied to $Set1$. The results represent the $WCET_{est}$ reduction if memory content selection is applied in combination with cache partitioning and multi-task program scratchpad allocation. The values in parantheses show the results from Table 4.3 if more than 0.1% lower $WCET_{est}$s are achieved compared to only applying *MPSPM* allocation and *CP*.

Since the maximum achievable $WCET_{est}$ reductions are identical to the results achieved by applying only *MPSPM* allocation and *CP*, the improvements by additional memory content selection are highlighted in the following. As already observed in Section 4.6.5.1, memory content selection yields an advantage for smaller cache sizes where not all code on the critical path does fit into the cache simultaneously. Up to 23.4% w.r.t. the $WCET_{est}$ of the task set can be saved for 10% cache size and 1% SPM size (note that the 0% row only reflect the results gathered in Section 4.6.5.3). *MCS* is able to improve the $WCET_{est}$ for smaller SPM sizes (1–3%) where not all hotspots can be moved to the fast SPM. If the cache and the

Table 4.6: Optimized $WCET_{est}$ for MPSPM allocation, CP and MCS applied to $Set2$

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	86.9% (100.0%)	83.0% (97.2%)	66.0% (91.9%)	60.2% (80.2%)	60.1% (77.5%)	52.6% (76.8%)	52.6% (69.1%)
1%	100.0%	86.9% (100.0%)	83.0% (97.2%)	66.0% (91.9%)	60.1% (80.2%)	60.1% (77.5%)	52.6% (76.8%)	52.6% (69.1%)
3%	100.0%	72.8% (83.5%)	66.0% (75.8%)	65.9% (70.3%)	60.1% (64.2%)	60.1% (60.5%)	52.6% (59.7%)	52.6% (56.3%)
5%	83.8%	65.9% (66.5%)	65.8%	64.7%	57.1%	57.1%	52.6%	52.6%
10%	70.3%	65.7%	65.7%	60.1%	52.5%	52.5%	52.2%	52.2%
15%	65.7%	62.4%	58.5%	53.3%	52.5%	52.5%	52.0%	52.0%
20%	60.4%	60.1%	53.8%	53.2%	52.5%	52.4%	52.1%	52.0%
25%	56.6%	56.6%	55.8%	53.2%	52.5%	52.5%	52.1%	52.1%

SPM grow in size, the number of cache conflict decreases as well as the performance of *MCS*.

The results for applying *MPSPM*, *CP* and *MCS* to $Set2$ are shown in Table 4.6. As already observed for $Set1$, *MCS* performs well for SPM sizes up to 3% of the overall task set size. Up to 25.9% lower $WCET_{est}$s can be achieved if MCS is additionally applied (5% cache, 1% SPM). Since the remaining $Set3$ and $Set4$ behave similar, a full discussion is omitted but all results can be looked up in Tables A.3 and A.4, Appendix.

The results presented in Section 4.6.5.1 – 4.6.5.3 attest that adjusting and combining WCET-driven optimizations yields high optimization potential. However, in case of memory-based optimizations, an exact knowledge and consideration of the applied code modifications is mandatory to adjust the utilized optimizations.

4.6.5.4 Optimization Time

Evaluations performed in this chapter already pointed out that the optimization time of the presented WCET-driven optimizations highly depends on the complexity of the benchmarks to optimize as well as on the number of required WCET analyses. Since the optimization loop presented in Figure 4.15 often requires a high number of WCET analyses, this section evaluates the required optimization time.

Cache Partitioning combined with Memory Content Selection: The optimization time necessary for stand-alone cache partitioning was evaluated in Section 4.4.6.2. Now, it should be figured out how much this time is increased if memory content selection is coupled with cache partitioning.

Concerning cache-aware memory content selection, Equation (3.20), p. 67, states that for a task with n functions at most $3 + n$ WCET analyses are performed. According to Equation (4.7), p. 88, the number of WCET estimations during cache

Table 4.7: Optimization times for CP combined with MCS

Benchmark Set	Minimum	Maximum
$Set1$	25 min	96 min
$Set2$	47 min	103 min
$Set3$	29 min	191 min
$Set4$	52 min	120 min

partitioning depends on the number of tasks in set T and the number of considered cache partition sizes in set P. If cache partitioning is combined with memory content selection, the number of WCET analyses amounts to not more than:

$$\#Analyses_{WCET} = |P| * \sum_{i=0}^{|T|} n_i + 3 \tag{4.12}$$

This equation considers that the modified cache partitioning no longer has to perform WCET estimations since it can exploit the analysis results gathered during the memory content selection phase. Since the considered partition sizes grow with a power of two, the number of considered partition sizes $|P|$ grows logarithmic with the cache size.

Table 4.7 lists the minimum and maximum optimization runtimes for each task set if *CP* is combined with *MCS*. The minimum values represent the time required for 1% cache size compared to the overall program size, whereas the maximum values denote the times for 25% cache size. In the best case, 25 minutes are required to optimize $Set1$ whereas approximately 3 hours are spend in the worst case to optimize $Set3$.

Multi-task Program Scratchpad Allocation combined with Cache Partitioning: This paragraph evaluates the compilation time for a combined application of multi-task scratchpad allocation and cache partitioning to the benchmark sets. For the employed SPM allocation technique, the same as for cache partitioning applies: the optimization time is dominated by the number and complexity of performed WCET analyses.

During *MPSPM* allocation for a task set T, the number of analyses amounts to $2 * |T|$ (cf. Equation (4.11), p. 96). According to Equation (4.7), p. 88, the number of performed WCET analyses within *CP* for a set of partition sizes P amounts to $|T| * |P|$. For a combined optimization by both algorithms, the upper bound of performed WCET analyses computes as follows:

$$\#Analyses_{WCET} = |T| * (2 + |P|)$$

Since the number of considered partition sizes again depends on the available cache size, Table 4.8 depicts the minimum and maximum optimization time for each

Table 4.8: Optimization times for MPSPM allocation combined with CP

Benchmark Set	Minimum	Maximum
$Set1$	17 min	45 min
$Set2$	22 min	35 min
$Set3$	27 min	108 min
$Set4$	30 min	47 min

Table 4.9: Optimization times for MPSPM allocation combined with CP and MCS

Benchmark Set	Minimum	Maximum
$Set1$	26 min	93 min
$Set2$	42 min	114 min
$Set3$	28 min	216 min
$Set4$	60 min	151 min

task set. The shortest optimization time was observed for $Set1$ with 17 minutes whereas the longest time was consumed by $Set3$ with 108 minutes.

Multi-task Program Scratchpad Allocation combined with Cache Partitioning and Memory Content Selection: Finally, the required optimization time for applying all optimizations employed in this chapter should be evaluated. The *MPSPM* allocation performs $2 * |T|$ WCET analyses as computed in Equation (4.11), p. 96. The number of analyses for combined *CP* with *MCS* is computed according to Equation (4.12) such that the overall number of performed WCET estimations is not more than:

$$\#Analyses_{WCET} = 2 * |T| + |P| * \sum_{i=0}^{|T|} n_i + 3$$

Obviously, combining three optimizations exhibits the highest optimization time which is listed in Table 4.9. As for all combinations of optimizations with cache partitioning, the optimization time depends on the available cache size. In order to optimize $Set1$, at least 26 minutes are spent whereas at most four hours are required to optimize $Set3$.

The work presented in this chapter has not been published yet.

4.7 Summary

This chapter presented multi-task compiler extensions which were integrated into the WCC. Based on these extensions, novel WCET-driven optimization techniques for multi-task systems were presented. All optimizations are integrated into the WCC

compiler framework presented in Chapter 2. Unlike existing ACET and WCET optimizations which are able to apply code modifications to single tasks only, the presented optimizations exploit the introduced multi-task capabilities within WCC. Based on this framework, optimizations are able to consider the contribution of individual tasks to the overall system's WCET and the impact of modifications applied to one task on the remaining tasks. Furthermore, the combined application of a number of WCET-aware memory-based optimization was examined in order to point out interference and synergetic effects. Based on these results, the existing approaches were adapted in order to improve the optimization results. The effectiveness of the discussed optimizations exploiting the novel multi-task support was demonstrated on real-life benchmarks.

The first optimization proposed in this chapter was a technique to optimize multi-task systems equipped with caches. The technique of software based cache partitioning was exploited to improve the predictability of the worst-case cache behavior in focus of real-time systems executing multiple tasks. Employing partitioned caches, every task has its own cache area from which it cannot be evicted. The novel algorithm for WCET-aware software based cache partitioning in multi-task systems thereby achieves predictability of cache behavior. An ILP model, based on the tasks' WCETs for different partition sizes, selects the optimal partition size for each task with the objective of minimizing the system's WCET. The new technique was compared to existing partition size selection algorithms. Inspecting small task sets, the WCET is decreased by up to 30% compared to the standard approach. Better results can be achieved for larger task sets with up to 33% WCET reduction. On average, the size-based algorithm is outperformed by 12% for 5 tasks in a set, 16% for task sets with 10 tasks, and 19% considering tasks sets with 15 tasks.

The second optimization presented in this chapter targets the allocation of program code to scratchpad memories in a multi-task scenario. An existing scratchpad allocation technique for single tasks was extended in order to be aware of the tasks' WCETs and their contribution to the overall system WCET. Therefore, three heuristics were proposed which determine the share in SPM for each task based on static features. The first, WCET-based heuristic determines a task's scratchpad size based on its contribution to the system's WCET in a hyperperiod whereas the code size based heuristic determines the SPM size based on the tasks' code size. A hybrid heuristic combines the ideas of the last two in order to find the best compromise. It turned out that the WCET-based heuristic performs best in the majority of cases. Nevertheless, the hybrid heuristic was able to achieve the highest WCET reduction of up to 47.5%.

As third, an approach for memory architecture aware compilation was proposed. Therefore, existing WCET-directed optimizations were analyzed w. r. t. their influence on the memory access patterns of tasks in order to recognize possible interference. Based on these insights, a strategy was elaborated for a combined application of three optimizations presented in this thesis. A promising application order was proposed for the optimizations multi-task scratchpad memory allocation, software-

based cache partitioning and cache-aware memory content selection. Afterwards, the required modifications were presented which ensure a perfect cooperation. An evaluation attests that a necessarily increased optimization time pays off: WCET reductions of up to 48.3% for large caches and SPMs can be achieved by an intelligently combined application of the optimizations. However, the actual strength of the proposed memory architecture aware compilation is the optimization of systems with limited caches and SPMs. Significant improvements compared to separate application of the employed optimizations were achieved.

This chapter presents results indicating that common optimizations focusing on single tasks are not able to exploit the full optimization potential. In order to optimize a multi-task system, the full set of tasks has to be considered during the code optimization process. In case of WCET optimization, the resulting WCET data of all tasks and their execution counts within a hyperperiod have to be taken into account. Otherwise, a limited view of a single task may hide the influence of modifications of one task on the WCET of the remaining tasks.

Once again, this chapter demonstrated that considering the underlying hardware platform during optimization can yield the best performance by exploiting specialized hardware features. Likewise, tuning existing memory-based optimization for a combined application pays off in terms of WCET reduction.

Chapter 5

Compilation and Optimization for Multiple Targets

Contents

5.1 Introduction . . . 114
5.2 Related Work . . . 115
5.3 Retargetable WCET-aware Compiler Framework . . . 117
5.4 WCC Extensions . . . 118
5.4.1 Employing GCC for Code Selection . . . 118
5.4.2 General-Purpose Exchange of Information . . . 120
5.4.3 Retargetable Timing Analysis . . . 121
5.5 WCET-aware Static Locking of Instruction Caches . . . 122
5.5.1 Motivating Example . . . 123
5.5.2 Related Work . . . 125
5.5.3 Function-based static I-Cache Locking . . . 126
5.5.4 ILP-based static I-Cache Locking . . . 128
5.5.4.1 Lockdown Constraints . . . 129
5.5.4.2 Basic Block Costs . . . 130
5.5.4.3 ILP Model of the Control Flow of Functions . . . 130
5.5.4.4 ILP Model of the Global Control Flow . . . 131
5.5.4.5 Objective Function . . . 131
5.5.5 Evaluation . . . 132
5.5.5.1 Experimental Environment . . . 132
5.5.5.2 WCET Estimations . . . 133
5.5.5.3 Optimization Time . . . 135
5.5.5.4 Comparison with existing optimizations . . . 135
5.6 Summary . . . 137

5.1 Introduction

In the course of this thesis, a number of memory-based optimizations were presented and it was shown how code for multi-task systems can be compiled and optimized appropriately. All presented techniques have in common that they target a single architecture namely the TriCore TC1796. Other WCET-aware compiler frameworks (cf. Section 2.1) also model single processors. This strict binding to a particular hardware involves two disadvantages. First, any developed WCET-aware optimizations are exclusively designed for the supported processor. Since evaluations are limited to this processor as well, assumptions about the effectiveness of the optimization w. r. t. other processors are difficult or even impossible. A later evaluation might even reveal that an optimization performing well on the processor used during the optimization's design has a negative impact on another processor.

Second, a WCET-aware compiler framework that only supports a single processor does not benefit from synergies in the development of WCET-aware optimizations. Each generic software module implemented to assist optimizations for a particular processor must be partially or even completely rewritten when ported to another WCET-aware compiler framework. In contrast, a framework producing code for multiple processors might significantly shorten the development time of new processor-specific optimizations since generic modules can be reused. For example, data structures holding WCET information for particular code fragments might be developed for scratchpad optimizations for one processor and reused for cache optimizations of another processor.

Developing or porting optimizations to different target architectures is an extremely complex task: the employed compiler has to be ported to the new architecture which is equivalent to a completely new implementation of the code selector as well as the compiler backend. Writing a new code selector which generates small but efficient machine code is an error-prone and time-consuming task which should be avoided.

Standard compiler toolchains such as the *GNU Compiler Collection* (GCC) exist for a variety of target architectures but are limited to average-case optimizations. Since they lack a detailed WCET timing model and a multi-task support as provided by WCC (cf. Chapter 2, 4), such compilers are ineligible as basis for the development of WCET-driven optimizations. To overcome these obstacles, this chapter presents techniques which enable a straightforward integration of new target architectures into WCC. Therefore, a standard compiler is exploited as code selector and integrated into WCC's compilation flow. In this way, only the retargetable compiler backend has to be adapted to the new target architecture. The transformation of *flow facts* from the C source level into the compiler backend and WCET timing data in reverse direction is accomplished by a novel module exploiting debug information.

WCC's retargetability is demonstrated by exemplarily implementing support for the ARM architecture. Furthermore, a novel optimization for static locking of instruction caches is presented which exploits this ARM support in order to demon-

strate its suitability for use. The WCET-driven cache locking algorithm selects content to lock into the cache based on its influence on the WCET of a program to optimize. *Integer-linear programming* is employed to select the optimal set of memory blocks to lock and to optimize along the WCEP of a program.

The remainder of this chapter is organized as follows: Section 5.2 provides a brief overview of related work while Section 5.3 introduces the idea of a retargetable WCET-aware compiler framework. Multi-target extensions implemented into the WCC compiler are presented in Section 5.4. In Section 5.5, the static locking of instruction caches for the ARM architecture is proposed. Finally, this chapter is summarized in Section 5.6.

5.2 Related Work

One of the first works considering compilation for different target architectures dates back to 1958 and was published by Convay [24]. His *universal computer oriented language* (UNCOL) is an universal intermediate representation for compilers. For each programming language, a translator should translate a program to UNCOL. Thus, only a new compiler backend should to be developed to generate code for a new machine architecture. UNCOL has never been fully implemented since it was not possible to represent the demands of all programming language, target machines and their instruction sets in a single language.

In contrast to earlier works on compilers, Glanville [43] focuses on retargetable code selection for a given, fixed programming language. A target-independent code generation algorithm emits code for a desired architecture by exploiting a *target machine description*. Nevertheless, the approach does not consider or support any form of compiler optimizations which are mandatory in order to achieve high code quality.

AT&T's retargetable compiler project `lcc` was examined by Fraser et al. in [40]. A C-specific, target-independent compiler frontend is employed. The retargetability of the code selector is driven by tables and a tree-grammar which map high-level constructs to instructions. Implementations for MIPS, SPARC and x86 are provided as examples and demonstrate the realization of simple low-level optimizations.

The most prominent retargetable compiler is the GNU Compiler Collection [41]. GCC is a perfect example how a compiler can be ported to a large number of programming languages, targets and host architectures. Therefore, the concept of coupling a language-configurable front end with an architecture-independent middle-end and a retargetable, target-specific backend is used. Optimizations can be applied at each abstraction level.

All approaches have in common that they do not explicitly consider code generation for embedded/cyber-physical systems and are not aware of highly specialized hardware features such as scratchpad memories or cache locking features.

In contrast, Marwedel et al. present various approaches and techniques for code generation targeting embedded systems [83]. Challenges in code generation and retargetability are treated as well as the exploitation of specialized instruction sets such as for DSP processors.

Leupers et al. achieve retargetability with their MSSQ compiler [69]. A hardware description language (HDL) specifies a target processor model as RT-level net lists. Based on this description, a code generator for the desired architecture is created fully automatically. Thereby, MSSQ can be ported to a number of architectures for which a HDL processor models is available with small effort. By implication, this means that porting MSSQ to commercially available processors is in the majority of cases impossible since the required HDL models are kept under wraps.

Parallel to the development of compilers translating a programming language into machine code, compiler for microprogramming emerged in the early 1980's. Such compilers translate programs into sequences of microinstructions which control the CPU on a fundamental level. In this way, single hardware circuits inside the CPU core can be driven such as connecting of certain registers to the *arithmetic logic unit* (ALU) or performing ALU operations. Baba et al. developed their own *microarchitectural programming language* called *MPGL* [7]. The corresponding retargetable compiler for microprogramming, called *MPG*, is based on a machine description which is exploited to translate MPGL statements to microinstructions. Although the generated code was efficient compared to hand-crafted microprograms, MPGL was not able to achieve acceptance.

Marwedel developed the *machine independent microprogramming language* (MIMOLA) and presented an appropriate retargetable microcode compiler as *MIMOLA Software System* [80]. The machine independent high-level microprogramming language uses a Pascal like syntax. Simultaneously, the language is employed for the hardware description required for the retargetable backend. In [81], Marwedel presents an extension called TREEMOLA which is employed to write structural hardware descriptions. In this way, a mapper can emit binary machine code without any instruction set description. Both [80] and [81] support horizontal microprograms which inherently support hardware parallel execution. Since [81] employs pattern matching between hardware structures and the program to compile, the code generation requires high computational power leading to high compilation times.

Also based on MIMOLA, Leupers et al. encounter high compilation times with a retargetable microcode compiler employing a bootstrapping technique. Therefore, a MIMOLA hardware description is compiled to a machine code compiler for the processor's instruction set. This compiler can be employed to compile high-level programs to native machine instructions.

None of the approaches listed above integrates a WCET timing model. Generally, compiler-based WCET minimization is sparsely dealt within today's literature. All the more, no retargetable WCET-optimizing compiler is known to the best of the author's knowledge.

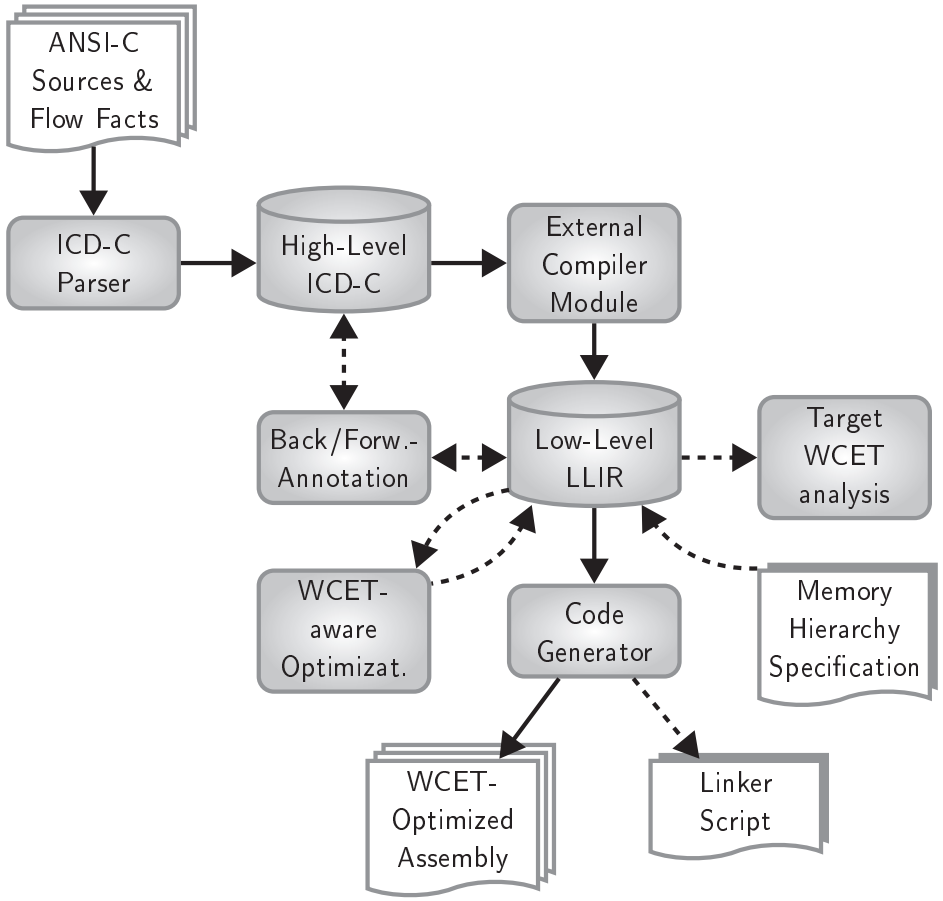


Figure 5.1: Workflow of the retargetable WCC

5.3 Retargetable WCET-aware Compiler Framework

This section introduces the techniques which extend WCC to a new architecture and shows the resulting compilation workflow. WCC's original compilation workflow was already presented in detail in Chapter 2, thus only modifications of the original workflow are discussed here. Figure 5.1 shows the compilation workflow as realized for the ARM architecture.

Basic components such as the compiler frontend ICD-C with its substantial set of high-level optimizations or WCC's memory hierarchy specification stay untouched. Reusing such core components eases the integration of new architectures. But the unavoidable implementation of a code selector for a new architecture is often a tedious task for which men-years can be spent. This task has to be done for every compiler in a similar fashion – regardless of the objective the compiler tries to optimize. Hence, the idea is to reuse existing implementations for code generation in order to port WCC to a new architecture.

To store the emitted assembly code, a target-specific implementation of the LLIR is required. Thanks to LLIR's modular structure and easy retargetability, implementing the target-specific parts is possible with justifiable expenditure.

As stated in Section 2.4.2, WCC's code selector is not only responsible for emitting efficient machine code. Furthermore, *flow facts* have to be transformed from their ICD-C representation to low-level equivalents attached to LLIR in order to enable an automated WCET analysis. To enable this transfer of information as well as annotating WCET data in the opposite direction, mapping information of ICD-C elements to LLIR structures has to be maintained. Standard compilers are not able to translate and maintain the information required for WCET analysis and optimization. Therefore, a module is proposed which supports backward/forward annotation of arbitrary data from one level of abstraction to the other if standard compilers are employed for code selection.

But even if *flow facts* can be transferred between both levels of abstraction, the interface to the static WCET analyzer has to be ported as well: up to now, the existing library (cf. Section 2.4.4.1) coupling the timing analyzer *aiT* is exclusively limited to the translation of TriCore assembly to an equivalent CRL2 representation. Thus, an interface for WCET estimations targeting arbitrary architectures has to be designed.

5.4 WCC Extensions

The overall workflow of the retargetable WCC was introduced in the previous section. In the following, a detailed discussion of the modified components is provided. How an arbitrary compiler can be employed for code selection within WCC is described in Section 5.4.1. A general-purpose module for exchanging information between WCC's abstraction levels ICD-C and LLIR is presented in Section 5.4.2. Finally, Section 5.4.3 sketches the platform-independent interface to the static WCET analyzer *aiT*.

5.4.1 Employing GCC for Code Selection

The first step towards a retargetable WCET-aware compiler framework is the elimination of the TriCore-specific code selector. This is achieved by substituting it by an arbitrary compiler for a processor supported by the static timing analyzer. The employed compiler should be considered as black box to enable easy updates to new versions or even the replacement by a totally different compiler for the same architecture. Thus, a modification of the employed compiler has to be avoided. This idea for employing a standard compiler for code selection was published in [94].

The approach presented in [94] is restricted to a direct compilation of C code generated out of an ICD-C IR to a binary program without access to the assembly code. Thus, no possibility of developing and applying assembly-level optimizations – as extensively done in this thesis – exists. In the following, a significantly extended

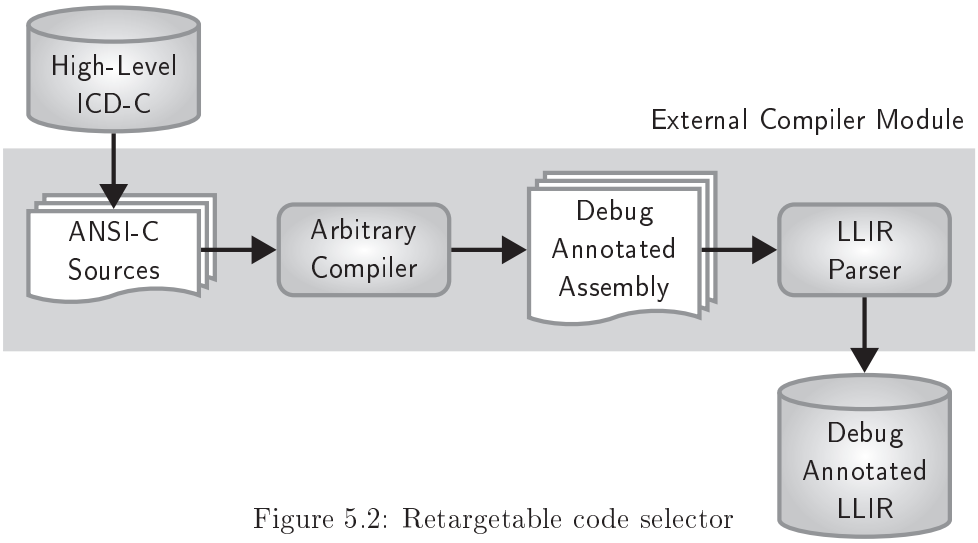


Figure 5.2: Retargetable code selector

approach is proposed which enables the manipulation of the generated assembly code and thereby supports the development of assembly-level optimizations in the same way as already available for the TriCore processor.

As depicted in Figure 5.2, the LLIR code selector (cf. Figure 2.2, p. 16) is replaced by an *external compiler module*. The input program, represented by an ICD-C IR, is emitted to equivalent C source files. Since ICD-C also serves as source-to-source optimizer, none of the analyses and optimizations provided by WCC's high-level IR is discarded but can be furthermore exploited. Due to WCC's flow fact manager, all source code *flow facts* are correctly adjusted and transformed during optimizations until dumping the IR into equivalent C code. In this way, *flow facts* stay valid if, for instance, *loop unrolling* is applied to the ICD-C IR, and the corresponding loop bounds are adjusted.

The dumped code is passed to an arbitrary compiler which generates equivalent assembly code for the target architecture. To support the backward/forward annotation presented in the following section, the invoked compiler is instructed to generate debug information. This information maps C statements to corresponding parts of the assembly code. Unfortunately, the quality of the mapping information depends on the complexity of the applied code modifications within the compiler. For example, if basic blocks are merged, newly created or deleted, a proper mapping to C statements cannot be ensured. To overcome this problem, the employed compiler is invoked without any optimizations which could modify the control flow and corrupt the attached debug information. Since WCC applies its own optimizations on both ICD-C and LLIR level, this limitation does not entail negative influence w. r. t. the performance of the generated code.

The generated assembly annotated with debug information is parsed and equivalent LLIR structures are created. Structures such as functions, basic blocks and instructions are represented in a similar fashion in assembly and LLIR. However,

assembly source files do not include explicit information on a program's control flow. In the LLIR, such information is expressed by predecessor/successor relations. Thus, this data has to be extracted by analyzing the last instruction of each basic block in order to determine its possible successors and link the blocks in the corresponding LLIR.

The techniques presented in this Section, pooled in the box labeled *External Compiler Module* in Figure 5.2, are a replacement for WCC's TriCore-specific *code selector* (cf. Section 2.3).

5.4.2 General-Purpose Exchange of Information

Inside WCC, an exchange of information between the two abstraction levels ICD-C and LLIR is essential: The concept of *flow facts* storing user annotations on loop bounds and recursion depths was already introduced in Section 2.4.5. Since these *flow facts* are mandatory for WCET estimations, they are always kept valid and consistent during optimizations and the code selection phase.

Exchange of information is required in the opposite direction, from LLIR to ICD-C, as well. WCC already provides a module named *Back-annotation*, a methodology enables transformation of objectives between both levels. The approach relies on the ICD-LLIR code selector to keep track of which ICD-C IR constructs correspond to which ICD-LLIR components. The mapping function with the finest granularity of basic blocks in both levels is defined as follows:

$$backann_b : bb_{low} \rightarrow bb_{high}$$

This eases the exchange of information between both code representations. When applying an arbitrary compiler without the insight into its code selector, the compiler must be considered as a black box where the relationship between the source code and the binary executable is not apparent. Thus, a concept has to be developed to gather a proper mapping of objects from one abstraction level to the other.

The problem of bridging the gap between the source code and the machine code is not new. Especially in the domain of software debugging, it is mandatory to analyze the binary executable by stepping through the source code. To inform the debugger, e. g. *GNU GDB*, which source code construct belongs to which machine code fragment, different debugging formats, e. g. *STABS* or *DWARF2* [26] (the latter is used in the framework presented in this chapter) are employed. In particular, problems arise when optimized code is debugged since an unambiguous mapping between the source code and the optimized target program becomes infeasible. This is known as the *Code Location Problem* [116]. The debug code is generated during the internal code selection phase of the compiler and inserted into the binary and is subsequently interpreted by the debugger. Since most compilers generate debug code before any code transformations are performed, it is crucial to disable all compiler optimizations to obtain a correct mapping of the original source code and the resulting machine code.

Since the involved compiler serves as a pure code selector in the proposed workflow, it produces non-optimized assembly annotated with *DWARF2* debug information. Among others, this information indicates which machine code results from which high-level constructs. The mapping function assigns assembly instructions to C statements specified by file, source code line and column:

$$\text{DWARF} : \textit{assembly instruction} \rightarrow \textit{source file}, \textit{line}, \textit{column}$$

With this information, it is straightforward to assign *FF* to such assembly basic blocks corresponding to a certain source level statement or – in the opposite direction – WCET information to the corresponding source code constructs. This mapping is performed by the WCC module labeled *backward/forward annotation* in Figure 5.1. Hereby, an automated WCET estimation is enabled by low-level *flow facts* and the derived WCET data can be employed to assist WCET-driven high-level optimizations.

5.4.3 Retargetable Timing Analysis

The static timing analyzer employed by WCC, AbsInt's *aiT*, supports a large number of processors. In combination with an appropriate compiler as code selector, the WCET-aware compiler should be highly flexible and allow the compilation and optimization for different target architectures. However, the initial interface coupling the static timing analyzer is limited to the TriCore architecture (cf. Section 2.4.4).

The module *LIBLLIRAIT* processes the LLIRs of a program to analyze and generates an equivalent binary representation in *aiT*'s CRL2 format. Since instructions and registers have to be created as CRL2 structures as well, the entire target instruction set has to be supported. Thus, the existing implementation would require a complete reimplementation for each additional architecture. Such an implementation is a tedious and error-prone task since the safeness of a WCET analysis depends on a proper representation of the program in CRL2.

As sketched in Figure 2.1, p. 14, *aiT* employs its own tool `exec2crl` which decodes a binary program and generates a CRL2 representation. This binary decoder is provided for all target architectures supported by *aiT* and ensures a perfect conversion to AbsInt's CRL2 format. Hence, `exec2crl` is exploited by WCC's retargetable timing analyzer interface. Therefore, the module *llirait* emits a binary executable which is provided as input to the WCET analyzer. The retargetable *aiT* interface is depicted in Figure 5.3.

Although *aiT*'s value and loop analysis extract loop bounds for simple loops, additional user annotations are usually required to make a WCET analysis feasible. A human-readable *AIS* file with user annotations is provided as input to `exec2crl`. Information attached to *flow facts* (cf. Section 2.4.5), the memory hierarchy specification (cf. Section 2.4.6) and the cache configuration can be specified:

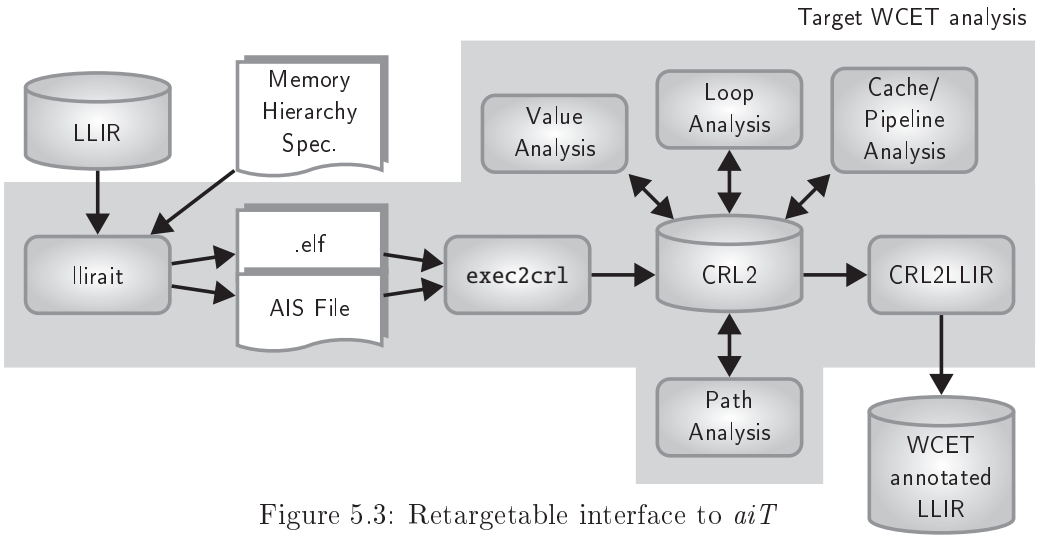


Figure 5.3: Retargetable interface to *aiT*

- Loop bounds
- Recursion depths
- Flow restrictions
- Memory areas with origin, size, timing, lockdown flag
- Cache size, line size, associativity, replacement policy

Now, the conventional *aiT* WCET analysis workflow starting with the binary decoder `exec2crl` as sketched in Figure 2.1, p. 14, is be employed for estimating the WCET of the program to analyze. Therefore, the binary program and the *AIS* file reflecting the program under analysis are provided as input. The programs belonging to *aiT*'s analysis flow as well as the input/output file handling are encapsulated in *LIBLLIRAIT* and are completely hidden from the user. This enables transparent and fully automated WCET analyses within WCC. The final CRL2 file is processed by the *CRL2LLIR* converter which extracts the determined WCET data. Finally, an LLIR annotated with WCET data is emitted.

The entire box labeled *Target WCET analysis* corresponds to the identically named box in Figure 2.1 and replaces the box labeled *aiT WCET analysis* of WCC's original compilation flow presented in Figure 2.2, p. 16. Hence, the approach proposed in this section can be transparently integrated into WCC's original workflow.

Apart from the idea of employing a standard compiler for code selection presented in [94], the work presented in this chapter has not been published yet.

5.5 WCET-aware Static Locking of Instruction Caches

Caches have become popular since they effectively improve the average-case performance. Nevertheless, they are a source of predictability problems due to their

```
1 void foo1(int x) {
2   if(x<100)
3     foo2();
4   else
5     foo3();
6 }
```

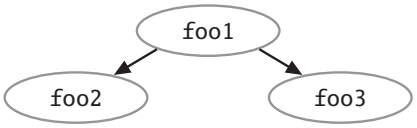


Figure 5.4: Exemplary program and resulting call graph

dynamic behavior. Various optimization techniques have been developed to improve a cache's predictability and improve their performance w. r. t. the runtime of a program. Examples are the *memory content selection* presented in Section 3.4 or the *cache partitioning* proposed in Section 4.4. Another cache-based optimization technique is *cache locking* which prevents cache content from eviction by manipulating the replacement strategy.

This section presents a novel WCET-aware optimization for static locking of instruction caches. The optimization aims at reducing the WCET of a program by statically locking parts of a program into the cache. Statically means that the cache content is loaded and locked before the program is executed and does not change during its execution. The proposed approach employs an *integer-linear program* to select memory blocks to lock. The ILP explicitly models the CFG of a program in order to cope with switching WCEPs. The impact of locked cache contents on the execution time of the contained basic blocks is modeled as well and thereby avoids repetitive WCET analyses.

The remainder of this section is organized as follows: Section 5.5.1 motivates the advantages of static cache locking by an example. An overview of related work considering memory and cache-based optimizations is provided in Section 5.5.2. Section 5.5.3 introduces a state-of-the-art, function-based instruction cache locking technique. The novel WCET-aware ILP-based algorithm for fine-grained static cache content selection is proposed in Section 5.5.4. An evaluation in Section 5.5.5 compares the performance of both the state-of-the-art cache locking techniques and the novel WCET-aware locking approach with a system equipped with a regular cache.

5.5.1 Motivating Example

This section demonstrates the advantages of static cache locking by an example. The amount of cache misses highly depends on the ratio of cache to memory size, the cache replacement policy and the structure of the executed program. A high amount of cache misses implies costly reloading of content from the slow main memory and leads to a high number of penalty cycles due to pipeline stalls.

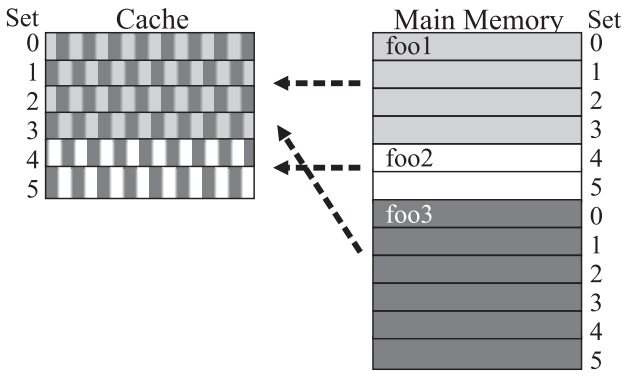


Figure 5.5: Worst-case cache behavior

Besides such unavoidable *real cache misses*, the computed WCET of a program is affected by the overestimation of a static WCET analyzer as well: if the memory address of an instruction fetch cannot be determined, it also cannot be determined if a memory access results in a cache hit or a cache miss. In such a case, the worst case – usually a cache miss – has to be assumed (called *assumed cache misses*). Figure 5.4 shows a code snippet and the resulting call graph for which such a situation can occur. If the value analysis of a static timing analyzer cannot determine whether `foo`'s parameter `x` is less than 100, a cache analysis has to consider both the `if`- and the `else`-path. For a memory layout as depicted in Figure 5.5, `foo1` and `foo2` are mapped to the same cache area as `foo3`. Now, the worst case has to be assumed where for each execution of `foo1`, `x` toggles between a value less than 100 and equal or above. Thus, for each execution of `foo1`, it has to be assumed that either `foo1` and `foo2` evict `foo3` from the cache or vice versa. This leads to an unnecessary high number of assumed cache misses if, for instance, `x` is usually below 100.

To overcome the problems of real and assumed cache misses, several processor architectures – especially in the embedded domain – are equipped with mechanisms to lock the content of caches. In this way, it can be ensured that an access to locked content always results in a cache hit and thus the number of real cache misses can be reduced. Overestimation of the WCET (caused by assumed cache misses) can be reduced as well since a static timing analyzer can doubtlessly determine which memory accesses result in cache hits and which ones have to be fetched from main memory.

Lockdown mechanisms with differing granularity exist. Either, each cache line can be locked individually or a way can be iteratively locked starting at the first line. However, different locking granularities and schemes are possible as well. For example, the embedded processor ARM926EJ-S [2], which is considered in this chapter, uses a cache-way-based locking scheme. This means that memory blocks with the cache's way size, starting at memory lines which are mapped to the first cache set, can be entirely loaded and locked into the cache. Dedicated locking bits steer if the normal cache allocation is allowed to access the corresponding cache way.

Selected content is loaded and locked into the cache by additional instructions located in the startup code of a program. This guarantees that the cache is filled and locked before the execution of the actual program starts.

5.5.2 Related Work

Falk et al. counteract possible predictability problems of caches with a static allocation of program code to so-called scratchpad memories (*SPM*) [32]. They employ *integer-linear programming* to select the optimal content of the SPM w. r. t. the program's WCET_{est}. The disadvantage of moving parts of the code to SPMs is the necessary correction of the control flow: far jumps have to be inserted to branch between different memories leading to an unavoidable code and runtime overhead. In contrast, the cache locking based optimization presented in this work exploits the transparent behavior of a cache and performs a lockdown of cache content inside the startup code of a program.

Another work considering scratchpad allocation is presented in [107]. Suhendra et al. developed an ILP-based allocation of frequently accessed data objects to faster memories in order to decrease the overall WCET_{est}. Their model of the program's WCET and possible execution paths serves as basis for the ILP-based algorithm presented in [32] and was also employed for the technique discussed in Section 5.5.4. Since only the intra-function control flow is modeled, a time consuming branch-and-bound approach or a sub-optimal heuristic is employed to optimize along the WCEP. Furthermore, moving data objects to SPM is much easier than locking instruction blocks into the I-cache. Data elements can be almost arbitrarily moved around to fill the SPM without gaps. Thereby, elements are never competing for the same memory/cache lines as occurring during cache locking.

In [42], Gebhard et al. present a technique for rearranging the positions of tasks to improve the cache performance. The interdependency relation of tasks is evaluated in order to determine a memory layout which maximizes the number of persistent cache sets for each task.

Papers [99] and [16] present techniques for statically locked instruction caches which are very close to the work presented in this paper. In [99], Puaut et al. present two algorithms which try to minimize the CPU utilization and the interferences between different tasks, respectively. Although they consider the WCET as metric, they are not able to react on switching WCEPs since they always optimize along an initially determined WCEP. Compared to [99], [16] presents an additional genetic algorithm which has the disadvantage that for each created individual, a time consuming WCET analysis has to be performed.

Puaut also presents techniques for I-cache locking [98] which consider changing WCEPs. However, the way how WCEPs are recomputed is not detailed. The authors use a parameter N trading off accuracy of WCEP recomputation with runtime consumption. Since runtimes for WCEP recomputation are still very high, the authors are unable to provide results for some of their benchmarks. In contrast, the

techniques presented in this work scale much better so that results for very large benchmarks can be gathered.

A function-based static I-cache locking is also proposed by the author of this thesis in [36]. An *Execution Flow Graph* (EFG) is employed to model possible execution paths on function level. The WCEP is determined by applying a modified Dijkstra algorithm before the most promising function on the WCEP is locked into the cache. To consider paths which are not the initial WCEP, two analyses are required for each alternative path in order to compute the gain of the functions on such a path. As opposed to this, the approach presented in this chapter only requires two WCET analyses in total. The work presented in [36] serves as reference implementation for function-based cache locking and is briefly introduced in Section 5.5.3.

An extension of the function-based locking presented in Section 5.5.3 was developed by Liu et al. in [72]. There, an *Execution Flow Tree* is presented which is traversed to generate a simple ILP which selects the functions to lock into the cache. In contrast to [72], the approach proposed in Section 5.5.4 is able to model the intra-function WCEP including loops at basic block level. The influence of the memory layout on lockable memory blocks – and thereby the runtime of basic blocks – is also taken into account; [72] ignores the fact that selected functions may conflict in the cache and thus cannot be locked simultaneously.

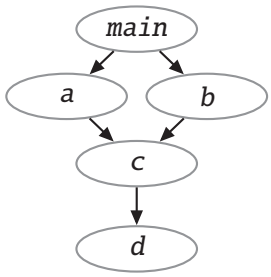
5.5.3 Function-based static I-Cache Locking

The choice of cache content to be locked has a significant impact on the WCET of a program. Due to the limited cache capacity, only content which highly profits from a cached execution should be locked. This section introduces an existing approach [36] which iteratively selects functions to be locked based on a so-called *execution flow graph* (EFG). The creation of the EFG, the WCEP construction as well as the I-cache content selection are described in the following.

The WCET analysis described in Section 2.2 considers different calling contexts for functions. Example 5.1 shows a conventional call graph where function `c` is called from `a` and `b`. The WCET analyzer *aiT* distinguishes two calling contexts for `c` and `d`. The first one is for the call from `a` and the resulting path `main`→`a`→`c`→`d`. The second one is for the call from `b` for the path `main`→`b`→`c`→`d`. Such contexts cannot be represented by means of a *call graph*.

Example 5.1 (*CG Contexts*)

For the following call graph, structural dependencies can be hidden within functions:



It cannot be distinguished if `main` calls `a` and `b` sequentially (code: `a(); b();`). Instead, they could be called mutually exclusive by `main` as well, resulting in the same *CG* (code: `if(x) a(); else b();`).

To model structural dependencies within functions as well as calling contexts which cannot be modeled using a CG, the concept of the *execution flow graph* has been developed:

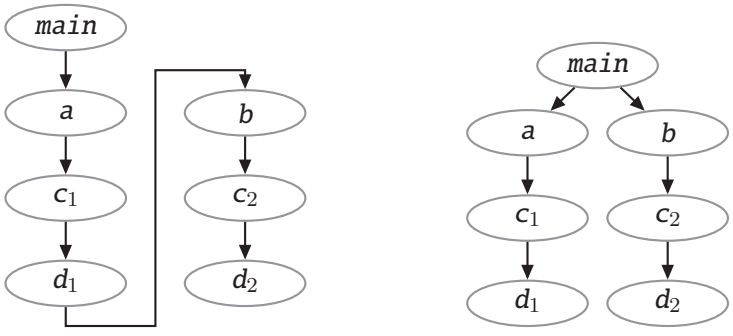
Definition 5.1

An *execution flow graph* is a weighted graph $G = (V, E, w_v, w_e)$. V represents the set of context-specific functions whereas $E \subseteq V \times V$ is a set of edges which connect two nodes: $(v_i, v_j) \in E$. The node weight w_v represents the context-specific WCET of a function v for a single execution of v. The edge weight $w_e = w(x, y)$ denotes an upper bound on how many times the execution flow passes from x to y in the considered context. The edges are created such that a path from the source node to the sink in the EFG corresponds to the ways of passing the control flow between functions.

A modified Dijkstra algorithm is applied to find all paths inside the *CG* in order to create the non-cyclical *EFG*. Unfortunately, such an explicit path enumeration can cause the *EFG* to grow exponentially with the size of the *CG*.

Example 5.2 (*EFG Creation*)

To transform the left *CG* in Example 5.1 into an *EFG*, the graph shown on the left-hand side below is created for the code `a(); b();` in `main`. If the code is `if(x) a(); else b();`, the following right-hand side graph is created:



> On the one hand, the graphs are able to model the structural dependencies within functions – here whether `b` is called in sequence with `a` or mutually exclusive. On the other hand, calling contexts of functions can be distinguished. Functions `c` and `d` both have two different contexts – one for the path `main`→`a`→`c`→`d` and one for the path `main`→`b`→`c`→`d`.

The *EFG* of a program is annotated with a profit value for each function f which is equal to the WCET reduction in cycles if f is locked into the cache. Therefore, the static timing analyzer *aiT* has to be invoked twice; once to determine the WCET if f is executed from main memory and once to determine the WCET if f is locked into the cache.

The function-based cache locking approach was not integrated into a sophisticated compiler framework with a tight coupling of a WCET analyzer. Without an interface to *aiT*'s CRL2 representation, only the WCET of functions on the WCEP can be determined. Thus, for each concurrent path, two separate WCET analyses have to be invoked in order to determine the WCETs of functions on this path.

To select functions to be locked into the cache, the function with the highest profit on the actual WCEP has to be determined. Since each leaf in the *EFG* represents the end of one path in the *CFG* of a program, only the profit for all nodes from a leaf to the root (usually the function `main`) have to be summed up to determine the length of the path. If the most promising function f is locked into the cache, its profit becomes zero for all calling contexts and the lengths of all paths including a call to f have to be recomputed.

5.5.4 ILP-based static I-Cache Locking

Locking cache content based on entire functions can have the disadvantage that unused code blocks occupy useful cache memory. Parts of locked functions which are not part of the actual WCEP do not contribute to the WCET but waste cache memory. Therefore, this section proposes an ILP-based cache content selection algorithm which is not limited to function boundaries but locks content based on the cache's way size.

For an n-way set-associative cache with a size of S_{cache} bytes and a line size of S_{line} bytes, each way consists of l cache lines:

$$l = \frac{S_{cache}}{n * S_{line}}$$

Each way comprises S_{way} bytes:

$$S_{way} = S_{cache}/n$$

Due to the modulo addressing function of cache controllers, memory addresses $addr$ mod $S_{way} = 0$ are mapped to the beginning of a cache way. Thus, the main memory can be divided into memory blocks $mb_1 ... mb_m$ with a size of S_{way} bytes (cf. Figure 5.6) for which each block can be entirely locked into a single cache way.

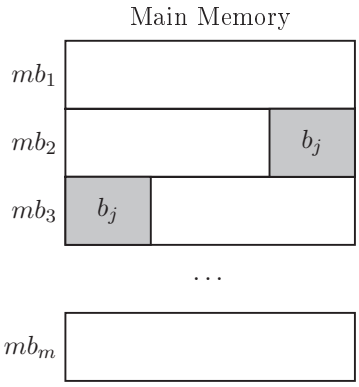


Figure 5.6: Memory layout divided into blocks with the cache's way size

For a way-based lockdown of cache content as supported by the ARM926EJ-S, only memory blocks with a granularity based on such a partitioning can be locked into the cache. Loading content from the main memory and locking it into a single cache line causes some costs C_{line}. Thus, the costs for locking a complete way consisting of L cache lines are as follows:

$$c_{lock} = L * C_{line} \tag{5.1}$$

In order to support an automatic lockdown of instruction caches to reduce the WCET, possible WCEP switches have to be recognized and handled which makes optimization challenging. The following sections present the novel ILP-based optimization technique which is capable of modeling a program's control flow and thereby ensures that optimizations are always performed along the WCEP. It determines an optimal set of memory blocks to lock into the cache w. r. t. the WCET of a program. Only *two* WCET analyses are required and the influence of locked cache lines on the execution time of a basic block is considered within the ILP. The following Section 5.5.4.1 models the costs for the locking of cache lines. Section 5.5.4.2 introduces basic block costs representing their execution times, whereas 5.5.4.3 describes the modeling of a function's control flow in the ILP. Afterwards, Section 5.5.4.4 models the global control flow whereas Section 5.5.4.5 describes the ILP's objective function.

5.5.4.1 Lockdown Constraints

In the following, ILP variables are represented using lowercase letters whereas constants are represented by uppercase letters. For ILP-based cache locking, the code of a program to be optimized has to be considered as m memory blocks with a size equal to the cache's way size S_{way}. The blocks start at memory addresses which are mapped exactly fitting into cache ways (cf. Section 5.5.4 as well as Figure 5.6).

For each of these blocks, a binary decision variable x_i decides whether memory block mb_i is locked into an arbitrary cache way:

$$x_i = \begin{cases} 1, & \text{if memory block } mb_i \text{ is locked into the cache} \\ 0, & \text{else} \end{cases} \tag{5.2}$$

An n-way set-associative cache can keep copies of up to n such memory blocks at the same time since ways can only be locked entirely. Thus, a constraint has to be formulated to ensure that the size of the content to lock does not exceed the cache size:

$$\sum_{i=1}^{m} x_i \leq n \tag{5.3}$$

5.5.4.2 Basic Block Costs

The time for a single execution of a basic block b_j is represented by C_j^{main} if b_j is entirely executed from main memory whereas C_j^{cache} is the execution time if the block is locked into the cache. s_j is the size of basic block b_j in Figure 5.6 in bytes and s_j^2 is the amount of bytes from basic block b_j overlapping with memory block mb_2. Then, p_j^2 is the runtime reduction in cycles if parts of basic block b_j located in memory block mb_2 are fetched from the cache due to a lockdown of mb_2. Generalized, the profit of basic block b_j for locking a memory block mb_i can be calculated as follows:

$$p_j^i = \frac{s_j^i}{s_j} * (C_j^{main} - C_j^{cache}) \tag{5.4}$$

Each basic block b_j of a function F causes some costs c_j. These costs represent the WCET of b_j depending on the memory from which b_j's instructions are fetched. If b_j or parts of it are locked into the cache, the execution time decreases by p_j^i cycles:

$$c_j = C_j^{main} - \sum_{i=1}^{m} x_i * p_j^i \tag{5.5}$$

5.5.4.3 ILP Model of the Control Flow of Functions

Modeling of control flow paths inside an ILP is performed similar to the modeling in Section 3.3.3.3. Thus, only a short recapitulation is provided at this point. For a basic block b_{exit}^L being the exit node of a loop L, its WCET w_{exit}^L is equal to its costs:

$$w_{exit}^L = c_{exit}^L$$

The WCET of a path leading from a node $b_j \neq b^L_{exit}$ in L to one of the exit nodes b^L_{exit} must be greater than or equal to the WCET of any successor b_{succ} of b_j in L, plus the costs c_j of b_j:

$$\forall b_j \in V \setminus \{b^L_{exit}\} : \ \forall (b_j, b_{succ}) \in E : \\ w_j \geq w_{succ} + c_j \tag{5.6}$$

In order to model multiple executions of L, all contained nodes are represented by a super-node v_L. The costs of v_L are the product of L's WCET for a single execution and L's maximal loop iteration count:

$$c_L = w^L_{entry} * Count^L_{max}$$

5.5.4.4 ILP Model of the Global Control Flow

The global control flow of a program is modeled within the ILP as follows: b^F_{entry} is assumed to be the entry basic block of function F. The ILP variable w^F_{entry} denotes the WCET of any path starting at b^F_{entry} for a single execution of F.

F's WCET represented by variable w^F_{entry} has to be added to the WCET of each block b_j calling F.

$$\forall b_j \in V \setminus \{b^L_{exit}\} : \ b_j \text{ calls } F : \ \forall (b_j, b_{succ}) \in E : \\ w_j \geq w_{succ} + c_j + w^F_{entry} \tag{5.7}$$

5.5.4.5 Objective Function

The overall goal of the ILP is to minimize a system's WCET by locking memory blocks into cache ways. Due to the nature of Equations (5.6) and (5.7), variable w^F_{entry} corresponds to the WCET of function F including the WCETs of all functions called by F. Function `main` is the unique entry point of an entire program; hence, variable $w^{\texttt{main}}_{entry}$ denotes the WCET of the program.

Since the cache has to be filled with code and locked in advance, overhead in the form of execution cycles arise before the program's execution. For a cache with ways composed of L cache lines, Equation (5.1) is extended to model the overall lockdown overhead:

$$o_{lock} = \sum_{i=1}^{m} x_i * L * C_{line} \tag{5.8}$$

This overhead has to be added to `main`'s WCET to obtain the overall WCET of a system:

$$w_{system} = w^{\texttt{main}}_{entry} + o_{lock} \tag{5.9}$$

Finally, the value of this variable has to be minimized by the ILP:

$$w_{system} \rightsquigarrow min. \tag{5.10}$$

5.5.5 Evaluation

In order to demonstrate the effectiveness of the novel WCET-aware cache locking technique, the approach is applied to a set of real-life benchmarks. In Section 5.5.5.1, the experimental environment is described which is employed to perform the evaluations. Section 5.5.5.2 discusses the WCET_{est} reductions achieved by our cache locking described in Section 5.5.4, whereas Section 5.5.5.3 discusses the required optimization runtime. Finally, Section 5.5.5.4 draws a comparison between our new optimization with the existing, function-based WCET-aware cache locking technique presented in Section 5.5.3.

5.5.5.1 Experimental Environment

For benchmarking, the ARM926EJ-S processor was employed which is equipped with 1 MB ROM as main memory from which content can be fetched within 6 cycles. The processor integrates a 16 kB I-cache with 32 bytes line size, least recently used (LRU) replacement strategy and a configurable associativity of 2 or 4. Content which is available in the cache can be accessed within 1 cycle whereas an access to the main memory constantly requires six cycles. The cache supports way-based cache locking for which Equations (5.1) – (5.10) are tailored. As stated in [36], loading and locking a single cache line of 32 bytes requires 47 cycles. These 47 cycles are used as constant C_{line} in Equations (5.1) and (5.8).

Uniformly, the optimization level *O3* is used for which the *WCC* compiler (cf. Figure 5.1, p. 117) applies 33 different optimizations in order to evaluate the performance of our new algorithms on highly optimized code. Refer to Section 2.4.7 for a detailed overview of applied optimizations.

For all evaluations, 100 benchmarks were used stemming from the benchmark suites *DSPStone* [130], *MediaBench* [65], *MRTC* [46] and *UTDSP* [115]. Additionally, a set of miscellaneous benchmarks was added referred to as *misc*. The code size of the benchmarks ranges from 100 bytes (*matrix_1x3*) up to 20 kB for the *gsm* benchmark.

As stated in Section 3.4.4, p. 65, the cache sizes are artificially limited to 10, 15 and 20% of the program's overall code size. This guarantees that a similar ratio of cache size to program size is used for all optimizations and static WCET analyses as found in current embedded systems. In this way, results which are comparable to the other cache and SPM-based optimizations presented in this thesis can be gathered. If the cache would be large enough to store large parts of the program, it would make more sense to use a scratchpad memory – which is fully predictable – instead. But for the sake of completeness, cache sizes of up to 100% of the program size are considered for a regular cache in order to explore the best possible reductions of the WCET_{est}.

For solving the ILP model generated by the algorithm in Section 5.5.4, *IBM ILOG CPLEX* [56] is utilized which is a sophisticated solver for *integer-linear programming* problems.

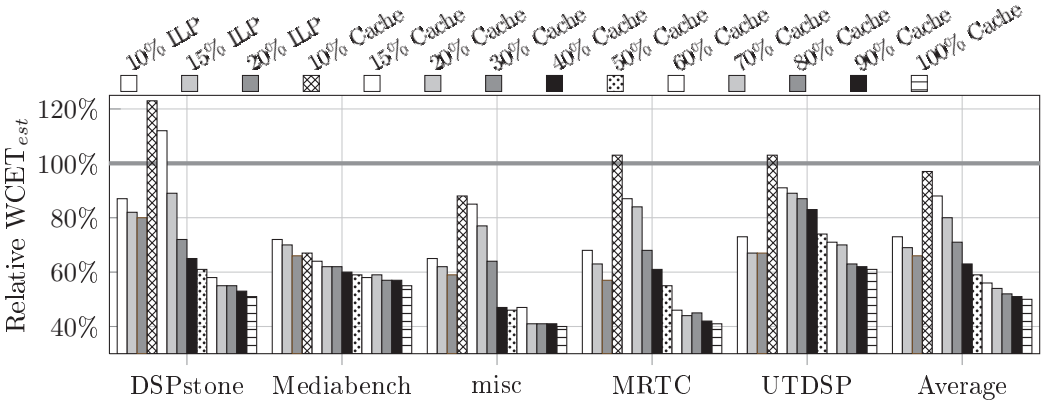


Figure 5.7: Relative WCET$_{est}$s for a 2-way set-associative cache

5.5.5.2 WCET Estimations

Figure 5.7 depicts the results achieved by the static cache locking algorithm if applied to the considered 100 benchmarks for a system equipped with a 2-way set-associative cache. The 100% reference line is equal to the estimated WCET of the benchmarks compiled with the optimization level *O3* executed in a system without any cache. For each benchmark suite, the left three bars represent the results achieved by the presented static cache locking technique if the cache amounts to 10, 15 and 20% of the overall program size. These resulting WCET$_{est}$s already include the overhead for loading and locking parts of a program's code into the cache before its execution. In order to assess the efficacy of static cache locking compared to a regular cache, the right bars represent the results if the benchmarks are executed on a system with a regular cache. All bars depict the average WCET$_{est}$ of the optimized programs of each benchmark suite computed by the static WCET analyzer as percentage of its "uncached" version.

By locking content into the I-cache, the ILP-based optimization is able to reduce the WCET$_{est}$ of the programs by up to 35.4% for 10% cache for the *misc* benchmarks. For the same benchmark set, the WCET$_{est}$ is reduced by up to 37.7% and 40.8% for 15% and 20% cache size, respectively. A system with a regular cache is able to achieve WCET$_{est}$ reductions of up to 32.7% for the *MediaBench* suite and 10% cache size. If the cache amounts to 15% and 20% of the overall program size, the WCET$_{est}$ is reduced by up to 36.2% and 37.8%, respectively.

On average over all considered 100 benchmarks, WCET$_{est}$ reductions of 27.1%, 31.2% and 34.3% can be achieved for 10%, 15% and 20% cache size if the novel ILP-based cache locking technique is applied. For a regular cache, however, only average WCET$_{est}$ reductions of 3.3%, 12.3% and 19.5% can be registered for 10%, 15% and 20% cache size. Here, the ILP-based cache locking optimization outperforms the regular cache by 23.8%, 18.9% and 14.8% for 10%, 15% and 20% cache size.

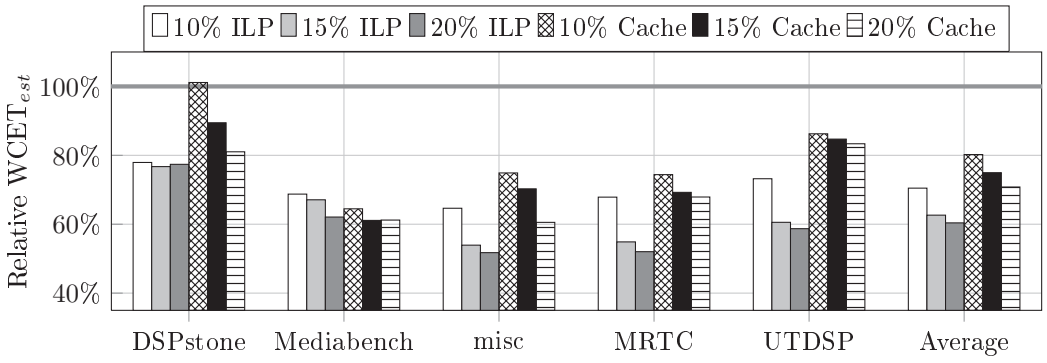


Figure 5.8: Relative WCET$_{est}$s for a 4-way set-associative cache

If the cache size of a system without cache locking is increased up to 100% of the program size, the WCET$_{est}$ is decreased by up to 59.7% for the *misc* benchmarks. Since the benchmarks which are gaplessly arranged in memory entirely fit into the cache, no cache misses due to evictions can occur. Thus, the achieved results represent the highest possible WCET$_{est}$ reductions for a regular cache as well as for a statically locked cache. On average, WCET$_{est}$ reductions of 50.3% for 100% cache size can be documented.

Although larger caches can keep copies of more instructions, it can happen that smaller caches achieve lower estimated WCETs due to less cache misses. For instance, a system executing the *MRTC* benchmark suite which is equipped with a cache of 70% size has a lower WCET$_{est}$ than with 80% size. This behavior is caused by the fact that by increasing the cache size, the mapping of memory blocks to cache lines is changed due to the modulo addressing function. Perhaps, other blocks now compete for the same cache lines resulting in a completely different eviction behavior and possibly increased WCET$_{est}$. For the cases where the capacity of a cache with l lines is doubled, consistently lower WCET$_{est}$s can be observed: the set of memory blocks mapped to the same cache line n is bisected into the blocks which are still mapped to the same line as before and the set of blocks which are mapped to cache line $l + n$. This can only lead to a reduction of cache misses which are induced by conflicts.

Even if static cache locking outperforms a normally operating cache in most of the cases, a normally operating cache performs better for the *MediaBench* suite. Since the content of a statically locked cache is not changed during the program's execution, the dynamic behavior of the cache gets lost. The benchmarks exhibit a number of computation kernels which cannot be locked simultaneously into the restricted cache. In contrast, the normally operating cache can exchange the content during execution of the program and always stores the kernel currently executed.

For caches with larger associativity, a memory block can be mapped to a larger amount of cache ways and thereby, the number of conflicts tends to be decreased. Figure 5.8 depicts the results if a 4-way set-associative cache is employed. Both

the system with a regular cache and a system with cache content locked by the optimization algorithm perform better than for a 2-way set-associative cache.

The novel ILP-based cache locking optimization is able to decrease the WCET_{est} compared to a system without cache by up to 35.4% for 10% cache size. For cache sizes of 15% and 20%, WCET_{est} reductions of up to 46.1% and 48.2% can be achieved for the *misc* benchmarks, respectively. Especially for small cache sizes of 10%, the regular cache significantly profits from a larger associativity and achieves WCET_{est} reductions of up to 35.5%. For caches with larger capacities of 15% and 20%, even higher WCET_{est} reductions of up to 38.9% and 38.8% are achieved, respectively.

Even though the regular cache outperforms the statically locked cache for *MediaBench*, the WCET_{est} reductions averaged over all benchmark suites are worse: The ILP-based cache locking decreases the WCET_{est} by 29.5%, 37.4% and 39.6% for 10%, 15% and 20% cache size, whereas the regular cache can only decrease the WCET_{est} by 19.8%, 25.0% and 29.2% for the same cache sizes.

5.5.5.3 Optimization Time

An Intel Xeon E5506 (2.13 GHz) was utilized to determine the time required for optimization of the benchmarks in Section 5.5.5.2. Most of the time necessary for WCET-aware static I-cache locking optimization was consumed by the WCET analyses using *aiT* which is always executed on a single CPU core.

For a single WCET analysis for a system without cache, up to 90 CPU minutes are required for the *latnrm_32_64* benchmark stemming from the *UTDSP* benchmark suite. Thereby, an optimization run spends up to 3 hours for the two required WCET analyses to determine the constants C_j^{main} and C_j^{cache} (cf. Section 5.5.4.2). But more than 90% of the considered benchmarks are analyzable within 2 minutes. This is still suitable for most application scenarios since the optimization essentially doubles the compilation time.

The complexity of solving the ILPs generated by the optimization discussed in Section 5.5.4 is of no practical relevance. For a CFG with n nodes, the ILP has a size of $O(n^2)$ constraints. For a program consisting of m memory blocks of size S_{way}, the ILP contains $O(n^2 + m)$ variables. The employed ILP solver *CPLEX* takes up to 1 CPU minute (*lmsfir_32_64* from *UTDSP*) but mostly terminates within a few seconds for the considered benchmarks. Compared to the WCET analyses required to determine the cost constants C_j^{main} and C_j^{cache} for each basic block, these values are negligible.

5.5.5.4 Comparison with existing optimizations

Besides the considerable performance gain compared to a regular cache, the algorithm presented in this chapter also outperforms state-of-the-art optimizations for instruction cache locking. The algorithm introduced in Section 5.5.3 as well as the optimization presented by Liu [72] are only able to lock complete functions. Their

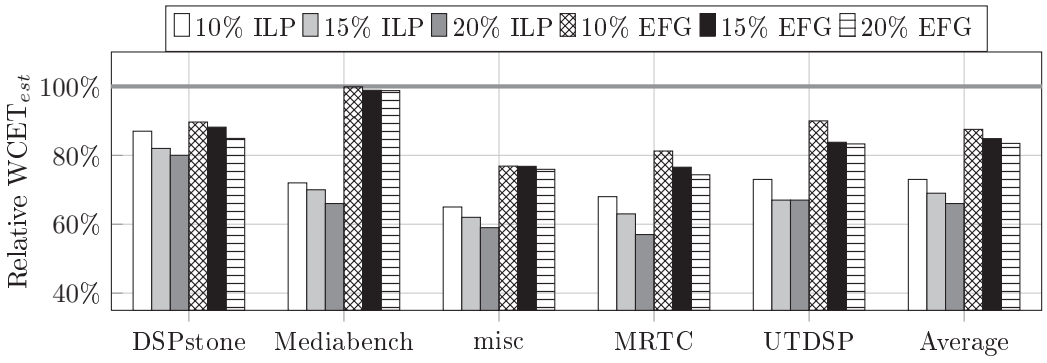


Figure 5.9: Relative WCET$_{est}$s compared to function-based locking

disadvantage is that if an entire function is locked into the cache, code blocks which are not part of the WCEP are locked into the cache as well. This wastes cache space and thereby optimization potential. Even so, it cannot be determined how much unused blocks are locked into the cache: for example, a block on the WCEP which eminently contributes to the WCET is locked into the cache. If, as a result, a WCEP switch occurs, it is possible that the locked block is no longer part of the new WCEP. Thus, only counting the locked instructions of the optimized program which are not part of the final WCEP cannot answer the question how many unused content is locked. Generally speaking, this observation applies for all cache locking based optimizations – also for the one presented in this chapter.

In the following, the ILP-based cache locking is compared to the *EFG* algorithm from Section 5.5.3 which is only marginally outperformed by Liu's improved approach. It turned out that function-based locking techniques are not well suited to handle small caches. If, for instance, cache sizes of 10% of the overall program size are considered, there are cases where no promising function fits into the cache. Thus, the function-based approach was extended such that if the most promising function does not fit into the remaining cache, only the beginning of this function is locked into the free cache memory.

Figure 5.9 shows the results achieved by applying the *EFG* algorithm compared to the ILP-based optimization presented in this section if a 2-way set-associative cache is considered. The new optimization outperforms locking techniques based on functions by up to 33% for the *MediaBench* suite. The *EFG* algorithm performs badly in this case, since most benchmarks have only few functions but monolithic computation kernels. The only locked functions do not entirely fit into the cache and their hotspots are often located in the middle or even at the end of the function which did not fit into the cache anymore.

But in most cases, the function-based locking profits from the applied modification which enables partial locking of promising functions into the remaining cache. Nevertheless, the function-based cache locking technique is outperformed by 14.6%, 16% and 17.7% w. r. t. WCET$_{est}$ reductions for 10%, 15% and 20% cache

size averaged over all considered benchmarks. This underlines the predominance of fine-grained cache locking techniques based on memory blocks instead of on entire functions.

Although an evaluation of the optimization runtimes was not performed, a qualitative statement can be made: to consider paths which are not the initial WCEP, the EFG approach requires two analyses for each alternative path in order to compute the gain of the functions on such a path. Since 97 of 100 considered benchmarks have concurrent paths, the amount of required WCET analyses and thereby the optimization runtime is at least doubled if the EFG algorithm is used compared to the ILP-based optimization.

The techniques and results presented in this chapter were published in [92].

5.6 Summary

This chapter proposed techniques making the development of retargetable WCET optimizing compilers possible. Based on WCC, extensions were developed which enable an easy porting of the WCET compiler framework to new platforms by exploiting existing standard compilers for the target architecture. The transformation of *flow facts* and WCET data between the abstraction levels was realized by an approach exploiting debug information of the employed compiler. Finally, timing estimations for arbitrary target platforms were made possible by a platform independent interface to the static WCET analyzer *aiT*. The WCC compiler was exemplarily ported to the ARM platform to demonstrate its capabilities.

Based on these extensions, a WCET-driven optimization technique for static instruction cache locking was presented. It was shown that the WCET_{est} of a program can be effectively reduced by locking parts of its code into the I-cache. The locked content is preserved against eviction leading to an increased number of cache hits and a decreased overestimation of its WCET.

Therefore, an ILP-based approach was presented which is tailored to select the content of instruction caches during compile time w. r. t. the WCET_{est} of a program to optimize. The selected content is loaded and locked by the startup code before the program's execution. The overhead for loading and locking code into the cache is explicitly considered in the ILP. Compared to existing approaches, repetitive time-consuming WCET analyses are avoided by modeling the control flow of the program in order to always optimize along a possibly switching WCEP.

It was shown that the novel cache locking algorithm was able to decrease the WCET_{est} of a set of real-life programs by up to 40.8%. On average over all considered benchmarks, WCET_{est} reductions between 27.1% and 39.6% were achieved for cache sizes ranging from 10% up to 20%. A regular cache is outperformed by 9.7% up to 23.8% and existing function-based cache locking techniques by 14.6% up to 17.7% for the same cache sizes, respectively.

This chapter demonstrated that a WCET compiler framework can be ported to other target architectures by exploiting standard components. Based on such an architecture support, the development of WCET-driven optimizations is fully supported with no additional effort. This is demonstrated by implementing an optimization which also underlines that exploiting the memory system of an underlying hardware platform can yield high WCET reductions. Furthermore, it was shown that synergies in the development of WCET-driven optimizations can be exploited; parts of WCC's TriCore-specific scratchpad memory allocation, were reused for the path modeling as part of the ILP of the static I-cache locking for the ARM platform.

Chapter 6

Summary and Future Work

Contents

6.1 Research Contribution of this Thesis 139
6.2 Future Work 142

In this thesis, the optimization potential of the memory subsystem of embedded/ cyber-physical systems used in the domain of real-time systems was examined. Approaches to reduce the WCET of embedded applications by optimizing their memory access patterns are presented. Evaluations underline the effectiveness of the techniques presented in this thesis and demonstrate the high optimization potential of real-time systems. All presented techniques allow an automated reduction of the worst-case execution time of a system and thus improve the current state of real-time system design. Up to now, well-established design methods were typically based on a manual and thereby time-consuming compilation and optimization process.

To enable the development of some of the optimizations integrated into WCC, the WCC framework was extended by capabilities for the compilation and optimization for multi-task systems. Furthermore, approaches were presented to retarget the WCET-aware compiler to new architectures. Finally, some of the introduced memory-based optimizations for single-task and multi-task systems underline the capabilities of the presented WCC extensions.

The contributions of the techniques presented in this thesis are summarized in Section 6.1. Finally, Section 6.2 concludes this thesis with a discussion on directions for future work.

6.1 Research Contribution of this Thesis

In the domain of real-time systems, the correctness of computations does not only depend on the results but also on the time interval within which these results can be delivered. The knowledge on an upper bound of a program's execution time, the WCET, is therefore mandatory. Based on the estimation of the WCET derived by static timing analyzers, it can be verified if tasks meet their deadlines.

With increasing computational power of processing units, memory subsystems have been identified as a performance-limiting factor. Common techniques which are employed to speedup memory accesses such as caches or branch prediction units

can have a hardly predictable or even adverse effect on the WCET. Whenever the actual behavior of the real hardware cannot be determined for a certain program point, a static timing analyzer has to assume the worst case. This leads to an undesired overestimation of the WCET.

This thesis introduces a number of optimizations which focus on increasing the efficiency of the memory subsystem of embedded real-time systems. On the one hand, the presented cache-based optimizations reduce the overestimation a static cache analysis introduces by reducing uncertainty w. r. t. the number of occurred cache hits and cache misses. On the other hand, all optimizations proposed in this thesis also achieve a real performance gain by speeding up memory accesses.

In the following, an overview on the proposed optimizations is provided in more detail – ordered by the closeness of the memories they are applied to. Starting within a CPU core, the instruction fetch buffer is the most tightly coupled memory which tries to provide the next few instructions to be executed. In this way, pipeline stalls where the CPU would otherwise wait for the completion of memory accesses should be avoided.

Branch Prediction aware Code Positioning, presented in Section 3.3, rearranges the order of basic blocks in order to reduce the number of mispredicted branches and to increase the utilization of the instruction fetch buffer. The number of unconditional jump instructions on the WCEP is reduced as well in order to decrease the WCET of a program. An evolutionary approach explores the space of achievable WCET reductions whereas an ILP-based approach achieves short optimization times at marginally worse results.

If a memory access fetching the next instruction cannot be served by the instruction fetch buffer, a main memory access is initiated. Caches are located between the CPU core and the main memory and store copies of recently used main memory blocks for a faster access. The efficiency of caches highly depends on the memory access pattern of the executed program.

Cache-aware Memory Content Selection, presented in Section 3.4, selects a subset of a program's functions to be cached. Only beneficial functions w. r. t. a program's WCET are placed into cached memories; less beneficial ones are moved to non-cached memory areas and thereby cannot cause cache evictions. A lower WCET is achieved by reducing the number of cache misses caused by mutual eviction of functions.

WCET-aware Static Locking of Instruction Caches, presented in Section 5.5, is another cache-based optimization. Memory blocks are loaded and locked into the cache before a program's execution in order to decrease the WCET by decreasing the number of cache misses caused by evictions. In this way, the overestimation of the WCET is reduced as well since a locked cache is fully predictable. An ILP-based

approach selects the memory blocks to lock and ensures an optimization along a possibly switching WCEP.

With growing demands on embedded/cyber-physical systems, the number of tasks running on such a system grows as well. This entails increasing demands on the memory subsystem since now a number of tasks compete for the same memories such as caches.

WCET-aware Software Based Cache Partitioning for Multi-Task Systems, presented in Section 4.4, is tailored to the optimization of multi-task sets. The instruction cache is divided into disjoint partitions, and each task is exclusively assigned to one of these partitions. One the one hand, predictability of the cache behavior is achieved since no inter-task cache interference can occur. But on the other hand, the performance w. r. t. the overall system's WCET is increased. Based on the task's contribution to the WCET within a hyperperiod, the optimal partition size is selected by an ILP-based approach.

However, caches always involve drawbacks in a real-time environment. Due to their autonomous controller logic, their behavior is not fully predictable. If the cache content is locked in order to eliminate such uncertainty, the controller logic becomes superfluous. Then, the occupied chip area causes unnecessary energy consumption and production costs. Therefore, scratchpad memories have been developed which exhibit the same access times as caches and are also tightly coupled to the CPU core.

WCET-driven Multi-Task Program Scratchpad Allocation, presented in Section 4.5, achieved WCET reductions by allocating code blocks of multi-task sets to scratchpad memories. An existing single-task PSPM allocation is extended by multi-task capabilities. Therefore, one of three heuristics can be employed which selects promising partition sizes for the tasks in a system based on different static features with the objective of WCET reduction. Afterwards, an existing PSPM allocation technique based on *integer-linear programming* performs the optimization for each task and the determined partition size.

All presented optimizations were developed and applied separately so far. A combined application of WCET-driven memory-based optimizations may be restricted in terms of optimization potential caused by mutual interfering modifications.

Memory Architecture aware Compilation, presented in Section 4.6, considers the impact of existing memory-based WCET optimizations on the memory layout and structure of a program. Based on an analysis of the optimizations' impact, an interference minimizing optimization order is proposed for a combined application. Furthermore, modifications are presented which exploit synergetic effects yielding better performance w. r. t. achieved WCET reductions than a separate application.

The presented optimizations are built in large part on infrastructure which cannot be found in common optimizing compilers. Even the WCET-aware C Compiler, introduced in Chapter 2, had to be substantially extended. Therefore, Section 4.3 introduces compiler extensions which enable an automated optimization of multi-task real-time systems. A representation of task sets with their representations at different levels of abstraction, scheduling parameters and an internal, common memory layout were developed.

Furthermore, techniques are introduced in Section 5.3 which add retargetability to the WCC compiler framework. The presented approach allows the compilation and optimization for new architectures. Porting of WCET-driven optimizations to new architectures as well as reusing existing data structures and optimization modules is also supported.

6.2 Future Work

The optimizations and compiler extensions presented in this thesis tackle a number of obstacles concerning the code generation and optimization for real-time systems. Nevertheless, there is always room left for improvements. In the following, ideas for future work are presented, grouped according to the corresponding chapters of this thesis.

WCC – WCET-aware C Compiler The initially single-task based WCC framework was extended by multi-task capabilities in the course of this work. However, embedded/cyber-physical systems are no longer limited to the execution of tasks on a single core. In recent times, multi-core systems are employed to satisfy demands for more and more computational power. Besides predictability problems coming along with multiple cores competing for the same resources such as level two caches or buses, no compiler support for multi-core compilation and optimization exists. Therefore, WCC should be extended by such capabilities in order to support the design of modern embedded/cyber-physical systems.

Another interesting perspective would be the consideration of the underlying operating system and its scheduling mechanisms during compilation. Up to now, task sets are optimized without having influence on the utilized scheduling policy or scheduling parameters which are treated as constants during the compilation process. Exchanging information between a WCET-optimizing compiler and the operating system could help to tighten the WCET estimation and optimize scheduling parameters for a lower system's WCET.

WCET-aware Memory-based Optimizations The memory-based optimizations (cf. Chapter 3) presented in this thesis optimize the performance of instruction fetches in different ways. Optimizations of data access in order to improve the data cache behavior or other parts of the memory subsystem is a promising research area.

Moreover, the presented branch prediction aware code positioning can be further improved if a modeling of pipeline dual and triple issues is integrated into the ILP.

Another extension is imaginable for cache-aware memory content selection. The granularity of code parts allocated to cached and non-cache memories could be refined if moving of code is not only applied on functions level. Instead, basic blocks as smallest unit should be considered.

Optimization of Multi-Task Systems The multi-task program scratchpad allocation presented in Chapter 4 achieves significant WCET reductions. Nevertheless, the optimization is based on heuristics which select a promising SPM partition size for each task without having knowledge on the actual impact on the WCET. A similar partition size selection as employed for the cache partitioning optimization could be beneficial. Therefore, SPM allocations for considered partition sizes should be performed and analyzed w. r. t. the resulting WCET. An ILP could select the optimal combination of partition sizes for the considered task set. To limit the optimization time, the employed single-task SPM allocation would have to be extended to break down the number of required WCET analyses which are responsible for the high optimization time.

Both the multi-task PSPM allocation and the cache partitioning are restricted to the optimization of program memory accesses. Since most embedded/cyber-physical systems are equipped with data SPMs and data caches, respectively, the optimizations should be extended to optimize data accesses as well.

Compilation and Optimization for Multiple Targets The presented support for multiple targets including compilation and WCET analysis creates a basis for comparing different architectures w. r. t. their computational power, code sizes or flexibility of instruction sets. Based on acquired experience concerning the influence of optimizations on different architectures, optimizations performing worse on a certain architecture could be either improved or excluded from the optimization process for this architecture.

Besides the support of multiple target architectures, the consideration of different objectives in the context of code generation for embedded/cyber-physical systems becomes more and more important. Thus, aiming at the optimization of single objectives such as WCET or ACET no longer sufficient. The resulting code size and energy consumption of such systems should be taken into account as well. Therefore, the WCC should be extended by a multi-objective support considering at least the objectives WCET, ACET, energy consumption and code size.

Finally, all optimizations presented in the course of this thesis could be extended to be aware of different objectives besides the WCET. In this way, the optimization of other objectives could be favored if the WCET of a task was reduced to the point that the task meets all deadlines.

Appendix A

Appendix

A.1 Definitions

A.1.1 Basic Block

Definition A.1

A *basic block* [88, p. 173] at assembly level is a sequence of instructions with maximum length which can only be entered by the first instruction and only be left by the last instruction. The first instruction is also referred to as *entry point* whereas the last instruction is also referred to as *exit point*

A.1.2 Control Flow Graph

Definition A.2

A *control flow graph* (*CFG*) is a directed graph representing all paths through a function F that might be traversed during its execution. The graph $G = (V, E, s)$ consists of a set of nodes V representing basic blocks and edges $E \subseteq V \times V$ connecting two nodes: $(v_i, v_j) \in V$ iff node v_j can be directly reached from v_i, thus v_j can be immediately executed after v_i. The entry node $s \in V$, called *source*, is the only node which has no incoming edges: $\nexists v \in V : (v, s) \in E$.

A.1.3 Interprocedural Control Flow Graph

Definition A.3

An *interprocedural control flow graph* (*IPCFG*) [119, p. 89–90] is the representation of an entire application by a directed graph. The graph $G = (V, E, s)$ is created by combining all CFGs $G_i = (V_i, E_i, s_i)$ of all n functions constituting a program:

- V: set of nodes, union of all node sets of all $G_i = (V_i, E_i, s_i)$
  $V = V_1 \cup V_2 \cup ... \cup V_n$

- E: set of edges, union of all edges of all $G_i = (V_i, E_i, s_i)$ combined with interprocedural control flow edges
  $E = E_1 \cup E_2 \cup ... \cup E_n \cup E^{CALL} \cup E^{RET}$

- $(v_i, s_j) \in E^{CALL}$: directed edge from the calling node v_i of the caller function to the source node s_j of the called function
- $(v_i, v_j) \in E^{RET}$: directed edge from the sink node v_i of the called function to the calling node v_i of the called function
- s: source node of the flow graph $G = (V, E)$ which is the entry node of the entire application (as a general rule the `main` function)

A.1.4 Knapsack problem

Definition A.4

Solving the *knapsack problem* means: Given n items, each with a certain benefit v_i and a weight w_i, determine the optimal combination of elements which does not exceed a limiting weight W but has a maximum overall benefit.

Formulated mathematically, the knapsack problem can be solved employing integer linear programming. A binary variable x_i decides whether item i should be placed in the knapsack:

$$x_i = \begin{cases} 1, & \text{if item } i \text{ should be placed in the knapsack} \\ 0, & \text{else} \end{cases}$$

A constraint is set up to limit the overall weight:

$$\sum_{i=1}^{n} x_i * w_i$$

The objective function representing the overall benefit has to be maximized:

$$\sum_{i=1}^{n} x_i * v_i \rightsquigarrow max.$$

A.2 Integer Linear Programming

When using Integer Linear Programming (ILP) to solve optimization problems, constraints of the following form have to be formulated:

$$\begin{aligned} a_{11}x_1 + a_{12}x2 &\leq b1 \\ a_{21}x_1 + a_{22}x2 &\leq b2 \\ a_{31}x_1 + a_{32}x2 &\leq b3 \end{aligned}$$

To support a sophisticated modeling of logical problem representations, the availability of further operators is required. In the following, operators and their modeling by ILP constraints are presented, which are employed in this work.

A.2.1 Not Equal Operator

To express that two variables a and b must not be equal, the following two constraints can be formulated, where x is a binary variable within an ILP and $C = \infty$ is a constant:

$$\begin{aligned} a &< b + C(1 - x) \\ b &< a + Cx \end{aligned} \tag{A.1}$$

A.2.2 Less-than Operator

Sometimes, it will be necessary to compare the value of two variables a and b and use the result as binary variable x within an ILP:

$$\begin{aligned} x &= (a < b) \\ &= \begin{cases} 1, & \text{if } a < b \\ 0, & \text{else} \end{cases} \end{aligned} \tag{A.2}$$

A constant $C = \infty$ is required to model the *less-than* operator which is mapped to binary variable x:

$$\begin{aligned} a &\leq b - 1 + C(1 - x) \\ a &\geq b - Cx \end{aligned}$$

A.2.3 Succeeding Operator

If the order of elements – e.g. basic blocks – should be modeled within an ILP, their absolute position within a sequence of elements is represented by their decision variable. To check whether two elements e_a and e_b are contiguously arranged, the *succeeding* operator evaluates their decision variables a and b, respectively. The result is mapped to binary variable x:

$$x = \begin{cases} 1, & \text{if } b \text{ directly succeeds } a \Leftrightarrow (a = b - 1) \\ 0, & \text{else} \Leftrightarrow (a \neq b - 1) \end{cases} \tag{A.3}$$

Again, the constant $C = \infty$ is used to enable the modeling of the *succeeding* operator. Two constraints are formulated which ensure that $x = 0$ for the case $a \neq b - 1$:

$$\begin{aligned} a - (b - 1) &<= C(1 - x) \\ (b - 1) - a &<= C(1 - x) \end{aligned}$$

A helper variable y is introduced to model two mutually exclusive constraints which ensure that $x = 1$ for the case $a = b - 1$:

$$a - (b - 1) < x + Cy$$
$$(b - 1) - a < x + C(1 - y)$$

A.2.4 AND Operator

ILP-based Optimizations may require complex conditions which have to be mapped to sets of constraints. In order to combine operators as the previously presented, a logical *AND* operator can be defined. The value of two binary decision variables a and b is evaluated and the result is mapped to a new binary variable x:

$$x = (a \vee b) \tag{A.4}$$

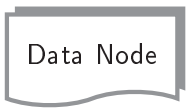
The necessary constraints are defined as follows:

$$x >= a + b - 1$$
$$x <= a$$
$$x <= b$$

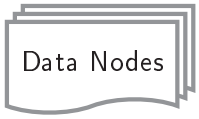
A.3 Flowchart Symbols

In the following, flowchart symbols used in this thesis should be explained.

A **Data node** represents input data such as a source file or a configuration file:

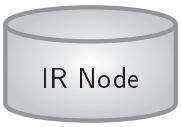


Collections of several data nodes are depicted as follows:



A **Tool Node** stands for a binary program or compiler module which performs some kind of processing:

An **IR Node** represents an instance of an intermediate representation (e.g. of ANSI C or assembly code):



A.4 Additional Results

This section depicts additional results which were gathered as part of the evaluation performed in Chapters 3 – 5.

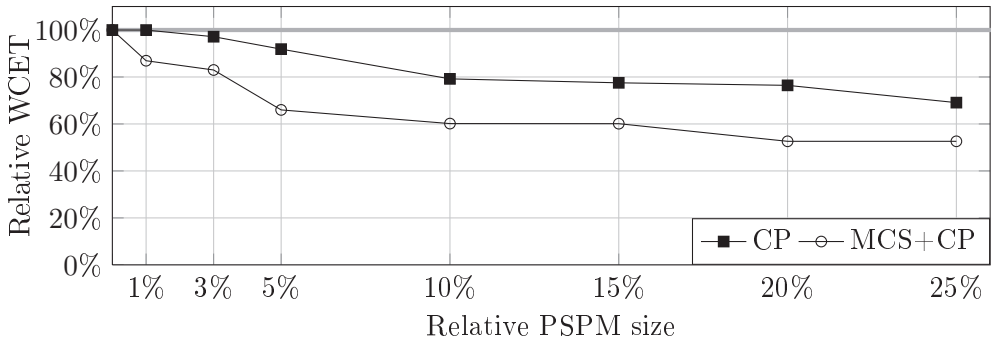


Figure A.1: Optimized WCET for cache-aware memory content selection and cache partitioning applied to *Set*2

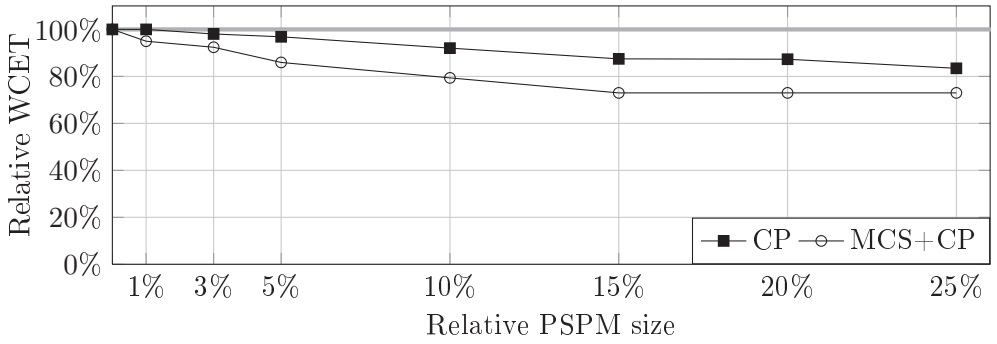


Figure A.2: Optimized WCET for cache-aware memory content selection and cache partitioning applied to *Set*4

Table A.1: Optimized WCET for MPSM allocation combined with CP applied to *Set*3

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	100.0%	100.0%	65.3%	57.2%	57.1%	56.0%	55.3%
1%	100.0%	100.0%	100.0%	57.6%	57.2%	57.0%	55.8%	55.0%
3%	57.1%	57.1%	57.1%	57.1%	56.6%	56.3%	55.6%	54.9%
5%	55.0%	55.0%	55.0%	54.7%	54.4%	54.1%	53.5%	55.0%
10%	54.4%	54.4%	54.4%	54.2%	53.8%	56.9%	53.0%	52.5%
15%	54.0%	54.0%	53.9%	53.7%	53.4%	53.3%	55.9%	55.0%
20%	54.0%	53.0%	53.6%	53.3%	52.4%	51.7%	52.1%	50.9%
25%	53.0%	53.0%	52.7%	52.7%	52.3%	51.7%	51.3%	51.1%

Table A.2: Optimized WCET for MPSM allocation combined with CP applied to *Set*4

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	100.0%	99.0%	96.9%	92.0%	87.5%	87.3%	83.4%
1%	100.0%	100.0%	99.0%	96.9%	92.1%	87.4%	87.3%	83.4%
3%	100.0%	100.0%	99.0%	96.9%	92.1%	86.8%	86.7%	83.2%
5%	90.7%	90.7%	87.7%	85.6%	82.9%	77.3%	77.1%	75.4%
10%	78.1%	76.7%	76.5%	74.5%	71.9%	67.0%	66.8%	66.0%
15%	72.2%	70.8%	70.7%	68.7%	66.1%	65.7%	65.5%	65.6%
20%	70.7%	70.7%	69.7%	68.2%	65.7%	65.6%	65.5%	65.4%
25%	69.4%	69.4%	66.2%	66.0%	65.7%	65.6%	65.4%	65.4%

Table A.3: Optimized WCET for MPSM allocation, CP and MCS applied to *Set*3

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	100.0%	99.5% (100.0%)	59.0% (65.3%)	57.2%	57.1%	56.0%	55.3%
1%	100.0%	99.9%	99.9%	57.5%	58.4%	58.4%	57.0%	57.0%
3%	57.1%	57.1%	56.5% (57.1%)	56.4% (57.1%)	55.8% (56.6%)	55.8% (56.3%)	55.1% (55.6%)	55.2%
5%	55.0%	55.0%	54.4% (55.0%)	54.3% (54.7%)	53.7% (54.4%)	53.7% (54.1%)	53.0% (53.5%)	53.0% (55.0%)
10%	54.4%	54.4%	54.0% (54.4%)	53.7% (54.2%)	53.1% (53.8%)	53.1% (56.9%)	52.4% (53.0%)	52.4%
15%	54.0%	54.0%	53.2% (53.9%)	53.2% (53.7%)	52.7% (53.4%)	52.7% (53.3%)	56.0%	52.0% (0.0%)
20%	54.0%	52.9%	52.5% (53.6%)	52.4% (53.3%)	51.5% (52.4%)	51.7%	52.1%	50.9%
25%	53.0%	53.9%	53.3%	53.0%	52.2%	51.7%	51.3%	51.1%

Table A.4: Optimized WCET for MPSM allocation, CP and MCS applied to *Set4*

SPM/C	0%	1%	3%	5%	10%	15%	20%	25%
0%	100.0%	95.0%	92.4%	85.9%	79.4%	73.0%	73.0%	73.0%
		(100.0%)	(99.0%)	(96.9%)	(92.0%)	(87.5%)	(87.3%)	(83.4%)
1%	100.0%	95.0%	92.4%	85.9%	79.4%	73.0%	73.0%	73.0%
		(100.0%)	(99.0%)	(96.9%)	(92.1%)	(87.4%)	(87.3%)	(83.4%)
3%	100.0%	95.0%	92.4%	86.0%	79.4%	72.7%	72.8%	72.8%
		(100.0%)	(99.0%)	(96.9%)	(92.1%)	(86.8%)	(86.7%)	(83.2%)
5%	90.7%	86.5%	84.0%	77.5%	72.0%	66.9%	66.9%	66.9%
		(90.7%)	(87.7%)	(85.6%)	(82.9%)	(77.3%)	(77.1%)	(75.4%)
10%	78.1%	75.5%	73.0%	71.3%	69.1%	65.8%	65.8%	65.8%
		(76.7%)	(76.5%)	(74.5%)	(71.9%)	(67.0%)	(66.8%)	
15%	72.2%	70.8%	70.8%	70.1%	67.2%	65.8%	65.8%	65.8%
20%	70.7%	70.7%	70.7%	68.6%	65.7%	65.6%	65.6%	65.6%
25%	69.4%	69.4%	68.0%	66.0%	65.7%	65.5%	65.5%	65.5%

List of Figures

1.1 Distribution of Execution Times 5
1.2 Example for WCEP variability 7

2.1 aiT WCET Analysis Workflow 14
2.2 Workflow of the WCET-aware C Compiler WCC 16
2.3 ICD-LLIR Class Hierarchy 19
2.4 Handling of LLIR Objectives 20
2.5 Coupling of aiT into WCC 21
2.6 Conversion of Flow Facts to CRL2 25
2.7 Infineon TriCore TC1796 Architecture [52] 28
2.8 Infineon TriCore TC1797 Architecture [55] 30

3.1 Rearranging code layout to support branch prediction 36
3.2 PISA Framework 40
3.3 Encoding of basic block positions. 41
3.4 One-point crossover reproduction 41
3.5 Typical Jump Scenarios 48
3.6 Relative $WCET_{est}$s after Code Positioning Optimizations 53
3.7 Relative ACETs after Code Positioning Optimizations 55
3.8 Exemplary Program and resulting Call Graph 59
3.9 Cache thrashed by mutual Evictions of Functions 59
3.10 Exploiting non-cached Memory Areas to avoid Cache Thrashing . . . 60
3.11 Relative $WCET_{est}$s after memory content selection 66

4.1 Workflow of the WCET-aware C compiler WCC with multi-task extensions 75
4.2 Class hierarchy of multi-task representation 77
4.3 Comparison between normally operating and partitioned cache . . . 78
4.4 Addressing of cache content 80
4.5 Mapping of tasks to cache lines 81
4.6 Optimized $WCET_{est}$ for DSPstone relative to standard approach . . 86
4.7 Optimized $WCET_{est}$ for MRTC relative to standard approach 87
4.8 Optimized $WCET_{est}$ for UTDSP relative to standard approach . . . 88
4.9 Scratchpad memory allocation problem 90
4.10 Optimized $WCET_{est}$ for multi-task PSPM allocation applied to $Set1$ 94
4.11 Optimized $WCET_{est}$ for multi-task PSPM allocation applied to $Set2$ 95
4.12 Optimized $WCET_{est}$ for multi-task PSPM allocation applied to $Set3$ 95
4.13 Optimized $WCET_{est}$ for multi-task PSPM allocation applied to $Set4$ 96
4.14 Memory layout interference 98

4.15 Schematic workflow of combined cache partitioning and memory content selection . . . 102
4.16 Optimized WCET for Cache-aware Memory Content Selection and Cache Partitioning applied to $Set1$ . . . 104
4.17 Optimized WCET_{est} for Cache-aware Memory Content Selection and Cache Partitioning applied to $Set3$ . . . 105

5.1 Workflow of the retargetable WCC . . . 117
5.2 Retargetable code selector . . . 119
5.3 Retargetable interface to aiT . . . 122
5.4 Exemplary program and resulting call graph . . . 123
5.5 Worst-case cache behavior . . . 124
5.6 Memory layout divided into blocks with the cache's way size . . . 129
5.7 Relative WCET_{est}s for a 2-way set-associative cache . . . 133
5.8 Relative WCET_{est}s for a 4-way set-associative cache . . . 134
5.9 Relative WCET_{est}s compared to function-based locking . . . 136

A.1 Optimized WCET for cache-aware memory content selection and cache partitioning applied to $Set2$ . . . 149
A.2 Optimized WCET for cache-aware memory content selection and cache partitioning applied to $Set4$ . . . 149

List of Tables

2.1 Available standard Compiler Optimizations 27

3.1 TriCore Jump Penalties 46
3.2 Benchmark Characteristics 52
3.3 Ratio of unconditional and mispredicted jumps 54
3.4 WCETs for Functions depending on the Memory Region 60
3.5 Benchmark Characteristics 65

4.1 Multi-task benchmark sets 93
4.2 Optimization times for multi-task SPM allocation 97
4.3 Optimized WCET_{est} for MPSPM allocation and CP applied to $Set1$ 105
4.4 Optimized WCET_{est} for MPSPM allocation and CP applied to $Set2$ 106
4.5 Optimized WCET_{est} for MPSPM allocation, CP and MCS applied to $Set1$ 107
4.6 Optimized WCET_{est} for MPSPM allocation, CP and MCS applied to $Set2$ 108
4.7 Optimization times for CP combined with MCS 109
4.8 Optimization times for MPSPM allocation combined with CP 110
4.9 Optimization times for MPSPM allocation combined with CP and MCS110

A.1 Optimized WCET for MPSM allocation combined with CP applied to $Set3$ 150
A.2 Optimized WCET for MPSM allocation combined with CP applied to $Set4$ 150
A.3 Optimized WCET for MPSM allocation, CP and MCS applied to $Set3$150
A.4 Optimized WCET for MPSM allocation, CP and MCS applied to $Set4$151

List of Algorithms

1 Greedy WCET-driven Memory Content Selection Algorithm 63
2 Pseudo code of cache partitioning algorithm 85

Bibliography

[1] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for TriCore, 2012. `http://www.absint.com/ait`. (Cited on pages 7, 12, 14 and 16.)

[2] Advanced RISC Machines Ltd (ARM™). *ARM926EJ-S Technical Reference Manual*, `ARM DDI 0198E` ed., 2001-2008. (Cited on page 124.)

[3] Almagor, L., Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S. W., Subramanian, D., Torczon, L., and Waterman, T. Finding Effective Compilation Sequences. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems, (LCTES)* (Washington, DC, USA, 2004), pp. 231–239. (Cited on page 99.)

[4] Altmeyer, S., Maiza, C., and Reineke, J. Resilience Analysis: Tightening the CRPD bound for set-associative caches. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* (Stockholm, Sweden, 2010), pp. 153–162. (Cited on page 74.)

[5] American National Standards Institute. Programming Languages - C, 2011. ISO/IEC 9899:2011. (Cited on page 4.)

[6] Austin, T., Larson, E., and Ernst, D. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer 35* (February 2002), 59–67. (Cited on page 13.)

[7] Baba, T., and Hagiwara, H. The MPG System: A Machine-Independent Efficient Microprogram Generator. *IEEE Transactions on Computers 30*, 6 (June 1981), 373–395. (Cited on page 116.)

[8] Beasley, J. E. *Advances in Linear and Integer Programming*. Oxford lecture series in mathematics and its applications. Clarendon Press, 1996. (Cited on page 16.)

[9] Bleuler, S., Laumanns, M., Thiele, L., and Zitzler, E. PISA — A Platform and Programming Language Independent Interface for Search Algorithms. In *Proceedings of the Conference on Evolutionary Multi-Criterion Optimization (EMO)* (Faro, Portugal, 2003), Springer, pp. 494 – 508. (Cited on page 39.)

[10] Bodin, F., and Puaut, I. A WCET-oriented Static Branch Prediction Scheme for Real-Time Systems. In *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)* (Palma de Mallorca, Spain, 2005), pp. 33–40. (Cited on page 38.)

[11] Börjesson, H. Incorporating Worst Case Execution Time in a Commercial C-compiler. Master's thesis, Uppsala University, Uppsala, Sweden, 1996. (Cited on page 12.)

[12] Briere, D. Overview on Airbus Fly-by-Wire Status. In *Proceedings of Aeronautics Days* (Hamburg, Germany, 2011). EADS Airbus SA. (Cited on page 2.)

[13] Briggs, P. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, TX, USA, 1992. (Cited on page 18.)

[14] Burguière, C., Rochange, C., and Sainrat, P. A Case for Static Branch Prediction Modeling in Real-Time Systems. In *Proceedings of the Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (Hong Kong, China, 2005), pp. 33–38. (Cited on page 37.)

[15] Burguière, C., Reineke, J., and Altmeyer, S. Cache-related Preemption Delay Computation for Set-associative Caches: Pitfalls and Solutions. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis (WCET)* (Dublin, Ireland, 2009). (Cited on page 74.)

[16] Campoy, A. M., Puaut, I., Ivars, A. P., and Mataix, J. V. B. Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)* (Palma de Mallorca, Spain, 2005), pp. 49–56. (Cited on page 125.)

[17] Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'boyle, M. F. P., and Temam, O. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (San Jose, CA, USA, 2007), pp. 185–197. (Cited on page 11.)

[18] Cavazos, J., and O'Boyle, M. F. P. Automatic Tuning of Inlining Heuristics. In *Proceedings of the Conference on Supercomputing (SC)* (Cambridge, Massachusetts, USA, 2005), pp. 14–25. (Cited on page 11.)

[19] Chamberlain, S., and Taylor, I. L. *Using ld*, 2000. Version 2.11.90, `http://www.skyfree.org/linux/references/ld.pdf`. (Cited on page 81.)

[20] Chattopadhyay, S., and Roychoudhury, A. Unified Cache Modeling for WCET Analysis and Layout Optimizations. In *Proceedings of the Real-Time Systems Symposium (RTSS)* (Washington, DC, USA, 2009), pp. 47–56. (Cited on page 13.)

[21] Chen, W. Y., Chang, P. P., Conte, T. M., and Wen-mei W. Hwu. The Effect of Code Expanding Optimizations on Instruction Cache Design.

IEEE Transactions on Computers 42, 9 (1993), 1045–1057. (Cited on pages 33 and 79.)

[22] Chiou, D., Rudolph, L., Devadas, S., and Ang, B. S. Dynamic cache partitioning via columnization. In *Proceedings of the Design Automation Conference (DAC)* (Los Angeles, CA, USA, 2000). (Cited on pages 78 and 79.)

[23] Colin, A., and Puaut, I. A Modular & Retargetable Framework for Tree-Based WCET Analysis. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)* (Delft, Netherlands, 2001), pp. 37–44. (Cited on page 13.)

[24] Conway, M. E. Proposal for an UNCOL. *Communications of ACM 1*, 10 (1958), 5–8. (Cited on page 115.)

[25] Denning, P. J. The Locality Principle. *Communications of the ACM - Designing for the mobile device 48*, 7 (2005), 19–24. (Cited on page 67.)

[26] The DWARF Debugging Standard. `http://dwarfstd.org`, September 2008. (Cited on page 120.)

[27] Eckart, J., and Pyka, R. ICD-LLIR Low-Level Intermediate Representation. `http://www.icd.de/es/icd-llir`, 2012. Informatik Centrum Dortmund. (Cited on pages 17 and 18.)

[28] Eurofighter Jagdflugzeug GmbH. Eurofighter Typhoon - Technical Guide, Hallbergmoos, Germany, 2011. (Cited on page 2.)

[29] European Telecommunications Standards Institute. DECT Standard, 2005. RTR/DECT-000225. (Cited on page 2.)

[30] Eyerman, S., Eeckhout, L., and Smith, J. E. Studying Compiler Optimizations on Superscalar Processors through Interval Analysis. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)* (Gothenburg, Sweden, 2008), pp. 114–129. (Cited on page 11.)

[31] Falk, H. WCET-aware Register Allocation based on Graph Coloring. In *Proceedings of the Design Automation Conference (DAC)* (San Francisco, CA, USA, 2009), pp. 726–731. (Cited on pages 34, 56 and 98.)

[32] Falk, H., and Kleinsorge, J. C. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the Design Automation Conference (DAC)* (San Francisco, CA, USA, 2009), pp. 732–737. (Cited on pages 32, 57, 62, 89, 91, 96 and 125.)

[33] Falk, H., and Kotthaus, H. WCET-driven Cache-aware Code Positioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (Taipei, Taiwan, 2011), pp. 145–154. (Cited on pages 11 and 38.)

[34] Falk, H., and Lokuciejewski, P. A compiler framework for the reduction of worst-case execution times. *Journal on Real-Time Systems 46*, 2 (2010), 251–300. (Cited on pages 7, 12 and 16.)

[35] Falk, H., Lokuciejewski, P., and Theiling, H. Design of a WCET-Aware C Compiler. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)* (Seoul, Korea, 2006), pp. 121–126. (Cited on page 20.)

[36] Falk, H., Plazar, S., and Theiling, H. Compile Time Decided Instruction Cache Locking Using Worst-Case Execution Paths. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Salzburg, Austria, 2007), pp. 143–148. (Cited on pages 61, 126 and 132.)

[37] Falk, H., Schmitz, N., and Schmoll, F. WCET-aware Register Allocation based on Integer-Linear Programming. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)* (Porto, Portugal, 2011), pp. 13–22. (Cited on page 34.)

[38] Falk, H., Wagner, J., and Schaefer, A. Use of a Bit-true Data Flow Analysis for Processor-Specific Source Code Optimization. In *Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia)* (Seoul, Korea, 2006), pp. 133–138. (Cited on page 19.)

[39] Ferdinand, C., and Wilhelm, R. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems 17*, 2-3 (1999), 131–181. (Cited on page 32.)

[40] Fraser, C. W., and Hanson, D. R. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley, 1995. (Cited on page 115.)

[41] GNU Compiler Collection. `http://gcc.gnu.org/`. (Cited on page 115.)

[42] Gebhard, G., and Altmeyer, S. Optimal Task Placement to Improve Cache Performance. In *Proceedings of the Conference on Embedded Software (EMSOFT)* (Salzburg, Austria, 2007), pp. 259–268. (Cited on pages 37, 61, 73 and 125.)

[43] Glanville, R. S. *A machine independent algorithm for code generation and its use in retargetable compilers.* PhD thesis, University of California, Berkeley, USA, 1977. (Cited on page 115.)

[44] Goodwin, D. W., and Wilken, K. D. Optimal and Near-optimal Global Register Allocations using 0–1 Integer Programming. *Software – Practice & Experience 26* (1996), 929–965. (Cited on page 34.)

[45] Guillon, C., Rastello, F., Bidault, T., and Bouchez, F. Procedure Placement using Temporal-Ordering Information: dealing with Code Size Expansion. *Journal of Embedded Computing 1*, 4 (2005), 437–459. (Cited on pages 37 and 61.)

[46] Gustafsson, J., Betts, A., Ermedahl, A., and Lisper, B. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis (WCET)* (Brussels, Belgium, 2010), pp. 136–146. (Cited on pages 51, 65, 85 and 132.)

[47] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)* (Austin, TX, USA, 2001), pp. 3–14. (Cited on pages 51 and 65.)

[48] Haffa, R. P., and Patton, J. H. Analogues of Stealth, 2002. Northrop Grumman Corporation. (Cited on page 2.)

[49] Hardy, D., and Puaut, I. Predictable Code and Data Paging for Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)* (Prague, Czech Republic, 2008), pp. 266–275. (Cited on page 13.)

[50] Hardy, D., and Puaut, I. WCET Analysis of Multi-Level Non-Inclusive Set-Associative Instruction Caches. In *Proceedings of the Real-Time Systems Symposium (RTSS)* (Barcelona, Spain, 2008), pp. 456–466. (Cited on page 86.)

[51] Hennessy, J. L., and Patterson, D. A. *Computer Architecture - A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, 2003. (Cited on page 67.)

[52] Infineon Technologies AG. *TC1796 User's Manual*, v2.0 ed. Munich, Germany, 2007. (Cited on pages 28, 35 and 153.)

[53] Infineon Technologies AG. *TC1796 Errata Sheet*, rel. 2.0 ed. Munich, Germany, 2008. (Cited on page 27.)

[54] Infineon Technologies AG. *TC1797 Errata Sheet*, rel. 1.3 ed. Munich, Germany, 2009. (Cited on page 27.)

[55] Infineon Technologies AG. *TC1797 User's Manual*. Munich, Germany, 2009. (Cited on pages 28, 30 and 153.)

[56] International Business Machines Corporation (IBM). IBM ILOG CPLEX V12.1, 2009. (Cited on pages 51, 84 and 132.)

[57] International Organization for Standardization. Extended BNF, 1996. ISO/IEC 14977:1996. (Cited on page 23.)

[58] Ishizaki, K., Takeuchi, M., Kawachiya, K., Suganuma, T., Gohda, O., Inagaki, T., Koseki, A., Ogata, K., Kawahito, M., Yasue, T., Ogasawara, T., Onodera, T., Komatsu, H., and Nakatani, T. Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler. In *Proceedings of the Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)* (Anaheim, CA, USA, 2003), pp. 187–204. (Cited on page 99.)

[59] Kennedy, K., and Allen, J. R. *Optimizing Compilers for Modern Architectures. A Dependence-Based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. (Cited on page 4.)

[60] Kim, C., Chung, S., and Jhon, C. An Energy-Efficient Partitioned Instruction Cache Architecture for Embedded Processors. *IEICE - Transactions on Information and Systems E89-D*, 4 (2006), pp. 1450–1458. (Cited on page 79.)

[61] Kirner, R. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Technische Universität Wien, Vienna, Austria, May 2003. (Cited on page 22.)

[62] Kleinsorge, J. C. WCET-centric code allocation for scratchpad memories. Master's thesis, TU Dortmund University, Germany, September 2008. (Cited on page 50.)

[63] Kleinsorge, J. C., Falk, H., and Marwedel, P. A synergetic approach to accurate analysis of cache-related preemption delay. In *Proceedings of the Conference on Embedded Software (EMSOFT)* (Taipei, Taiwan, 2011), pp. 329–338. (Cited on page 74.)

[64] Kowarschik, M., and Weiss, C. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies* (2003), Springer, pp. 213–232. (Cited on pages 33 and 79.)

[65] Lee, C., Potkonjak, M., and Mangione-Smith, W. H. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of International Symposium on Microarchitecture (MICRO)* (Washington, DC, USA, 1997), pp. 330–335. (Cited on pages 51, 65 and 132.)

[66] Lee, C.-G., Hahn, J., Seo, Y.-M., Min, S. L., Ha, R., Hong, S., Park, C. Y., Lee, M., and Kim, C. S. Analysis of Cache-Related Preemption

Delay in Fixed-Priority Preemptive Scheduling. *IEEE Transactions on Computers 47*, 6 (1998), 700–713. (Cited on page 73.)

[67] Lee, E. A. Cyber Physical Systems: Design Challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley, 2008. (Cited on page 1.)

[68] Lee, H., von Dincklage, D., Diwan, A., and Moss, J. E. B. Understanding the behavior of compiler optimizations. *Software – Practice & Experience 36* (2006), 835–844. (Cited on page 4.)

[69] Leupers, R., and Marwedel, P. Retargetable Code Generation Based on Structural Processor Description. *Design Automation for Embedded Systems 3*, 1 (1998), 75–108. (Cited on page 116.)

[70] Li, X., Liang, Y., Mitra, T., and Roychoudury, A. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming 69*, 1-3 (2007), pp. 56–67. `http://www.comp.nus.edu.sg/~rpembed/chronos`. (Cited on page 13.)

[71] Li, Y., Suhendra, V., Liang, Y., Mitra, T., and Roychoudhury, A. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *Proceedings of the Real-Time Systems Symposium (RTSS)* (Washington, DC, USA, 2009), pp. 57–67. (Cited on page 13.)

[72] Liu, T., Li, M., and Xue, C. J. Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking. In *Proceedings of the Symposium on Real-Time and Embedded Technology and Applications (RTAS)* (San Francisco, CA, USA, 2009). (Cited on pages 126 and 135.)

[73] Liu, X., Zhang, J., Liang, K., Yang, Y., and Cheng, X. Basic-block Reordering Using Neural Networks. In *Proceedings of the Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilaTion (SMART)* (Ghent, Belgium, 2007), pp. 12–26. (Cited on page 37.)

[74] Lokuciejewsi, P., Falk, H., Marwedel, P., and Henrik, T. WCET-Driven, Code-Size Critical Procedure Cloning. In *Proceedings of the Workshop on Software and Compilers for Embedded Systems (SCOPES)* (Munich, Germany, 2008), pp. 21–30. (Cited on page 17.)

[75] Lokuciejewski, P. Design and Realization of Concepts for WCET Compiler Optimization. Diploma thesis, TU Dortmund University, Germany, December 2005. (Cited on page 20.)

[76] Lokuciejewski, P., Cordes, D., Falk, H., and Marwedel, P. A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models. In *Proceedings of the International Symposium*

on Code Generation and Optimization (CGO) (Seattle, Washington, USA, 2009), pp. 136–146. (Cited on page 22.)

[77] Lokuciejewski, P., Falk, H., and Marwedel, P. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)* (Prague, Czech Republic, 2008), pp. 321–330. (Cited on pages 61 and 62.)

[78] Lokuciejewski, P., Plazar, S., Falk, H., Marwedel, P., and Thiele, L. Approximating Pareto optimal compiler optimization sequences – a trade-off between WCET, ACET and code size. *Software: Practice and Experience 41*, 12 (2011), pp. 1437–1458. (Cited on page 99.)

[79] lp_solve reference guide. `http://lpsolve.sourceforge.net/5.5/`. v. 5.5.0.14. (Cited on page 84.)

[80] Marwedel, P. A Retargetabe Compiler for a High-Level Microprogramming Language. *ACM Sigmicro Newsletter 15*, 4 (1984). (Cited on page 116.)

[81] Marwedel, P. Tree-Based Mapping of Algorithms to Predefined Structures. In *International Conference on Computer Aided Design (ICCAD)* (Santa Clara, CA, USA, 1993). (Cited on page 116.)

[82] Marwedel, P. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*, 2nd ed. Springer, 2011. (Cited on pages 1, 3 and 73.)

[83] Marwedel, P., and Goossens, G., Eds. *Code Generation for Embedded Processors, Dagstuhl Workshop, 1994* (1995), Kluwer. (Cited on page 116.)

[84] McKee, S. A. Reflections on the Memory Wall. In *Proceedings of the Conference on Computing Frontiers (CF)* (Ischia, Italy, 2004), pp. 162–168. (Cited on page 7.)

[85] Memik, G., Mangione-Smith, W. H., and Hu, W. Netbench: A benchmarking suite for network processors. In *Proceedings of the Conference on Computer-Aided Design (ICCAD)* (San Jose, CA, USA, 2001), pp. 39–42. (Cited on page 65.)

[86] Mitra, T., and Roychoudhury, A. A Framework to Model Branch Prediction for Worst Case Execution Time Analysis. In *Proceedings of the Workshop on Worst Case Execution Time Analysis (WCET)* (Vienna, Austria, 2002), pp. 68–71. (Cited on page 37.)

[87] Molnos, A. M., Heijligers, M. J., Cotofana, S. D., and van Eijndhoven, J. T. Cache Partitioning Options for Compositional Multimedia Applications. In *Proceedings of the Workshop on Circuits, Systems and Signal Processing (ProRISC)* (Veldhoven, Netherlands, 2004). (Cited on pages 78 and 79.)

[88] Muchnick, S. S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., 1997. (Cited on pages 18, 19, 26, 33, 42 and 145.)

[89] Mueller, F. Compiler Support for Software-Based Cache Partitioning. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems (LCTES)* (New York, NY, USA, 1995), ACM. (Cited on pages 77, 78, 79, 81, 82 and 84.)

[90] Nolte, T. *Share-Driven Scheduling of Embedded Networks.* PhD thesis, Mälardalen University, Sweden, 2006. (Cited on page 2.)

[91] Panda, P. R., Catthoor, F., Dutt, N. D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., and Kjeldsberg, P. G. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 6* (2001), 149–206. (Cited on page 98.)

[92] Plazar, S., Falk, H., and Marwedel, P. WCET-aware Static Locking of Instruction Caches. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (San Jose, CA, USA, 2012). (Cited on pages 9 and 137.)

[93] Plazar, S., Kleinsorge, J. C., Falk, H., and Marwedel, P. WCET-driven Branch Prediction aware Code Positioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (Taipei, Taiwan, 2011), pp. 165–174. (Cited on pages 9, 11 and 57.)

[94] Plazar, S., Lokuciejewski, P., and Marwedel, P. A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations. In *Proceedings of the Real-Time Systems Symposium (RTSS) WiP* (Barcelona, Spain, 2008), pp. 49–52. (Cited on pages 9, 118 and 122.)

[95] Plazar, S., Lokuciejewski, P., and Marwedel, P. WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis (WCET)* (Dublin, Ireland, 2009). (Cited on pages 9 and 88.)

[96] Plazar, S., Lokuciejewski, P., and Marwedel, P. WCET-driven Cache-aware Memory Content Selection. In *Proceedings of the International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)* (Carmona, Spain, 2010), pp. 107–114. (Cited on pages 9 and 67.)

[97] Prantl, A., Schordan, M., and Knoop, J. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Proceedings of the*

Workshop on Worst-Case Execution Time Analysis (WCET) (Prague, Czech Republic, 2008), pp. 141–148. (Cited on page 13.)

[98] Puaut, I. WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)* (Dresden, Germany, July 2006). (Cited on page 125.)

[99] Puaut, I., and Decotigny, D. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proceedings of the Real-Time Systems Symposium (RTSS)* (Austin, TX, USA, 2002). (Cited on pages 61, 79 and 125.)

[100] Pyka, R., and Eckart, J. ICD-C Compiler Framework. `http://www.icd.de/es/icd-c`, 2012. Informatik Centrum Dortmund. (Cited on page 17.)

[101] Queva, M. Phase-ordering in optimizing compilers. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2007. (Cited on page 99.)

[102] Schulte, D. Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler (in German). Diploma thesis, TU Dortmund University, Germany, 2007. (Cited on pages 23 and 24.)

[103] Sprague, R. E. A western view of computer history. *Communications of the ACM 15* (1972), 686–692. (Cited on page 1.)

[104] Srikant, Y. N., and Shankar, P., Eds. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002. (Cited on page 99.)

[105] Steinke, S., Wehmeyer, L., Lee, B.-S., and Marwedel, P. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)* (Paris, France, 2002). (Cited on pages 33 and 34.)

[106] Suhendra, V., and Mitra, T. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proceedings of the Design Automation Conference (DAC)* (Anaheim, CA, USA, 2008). (Cited on pages 61, 79 and 84.)

[107] Suhendra, V., Mitra, T., Roychoudhury, A., and Chen, T. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the Real-Time Systems Symposium (RTSS)* (Miami, FL, USA, 2005), pp. 223–232. (Cited on pages 38, 45, 90, 91 and 125.)

[108] Suhendra, V., Roychoudhury, A., and Mitra, T. Scratchpad Allocation for Concurrent Embedded Software. In *Proceedings of the Conference on*

Hardware/Software Codesign and System Synthesis (CODES+ISSS) (Atlanta, GA, USA, 2008), pp. 37–42. (Cited on pages 32, 89 and 91.)

[109] Synopsys. CoMET, Virtual Prototyping Solution1, 2012. `http://www.synopsys.com`. (Cited on page 56.)

[110] Tatlock, Z., and Lerner, S. Bringing Extensibility to Verified Compilers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (Toronto, Ontario, Canada, 2010), pp. 111–121. (Cited on page 12.)

[111] The European Parliament. Regulation (EC) No 661/2009. 13 July 2009. (Cited on page 2.)

[112] Theiling, H. Control Flow Graphs For Real-Time Systems Analysis. Ph. D. Thesis, Universität des Saarlandes, Saarbrücken, Germany, 2002. (Cited on page 14.)

[113] Theiling, H., Ferdinand, C., and Wilhelm, R. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Journal of Real-Time Systems 18*, 2-3 (2000). (Cited on pages 58 and 61.)

[114] Touati, S.-A.-A., and Barthou, D. On the Decidability of Phase Ordering Problem in Optimizing Compilation. In *Proceedings of the Conference on Computing Frontiers (CF)* (Ischia, Italy, 2006), pp. 147–156. (Cited on page 99.)

[115] UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, 2012. (Cited on pages 51, 65, 85 and 132.)

[116] Venturini, H., Riss, F., Fernandez, J.-C., and Santana, M. A Fully-non-transparent Approach to the Code Location Problem. In *Proceedings of the Workshop on Software and Compilers for Embedded Systems (SCOPES)* (Munich, Germany, 2008), pp. 61–68. (Cited on page 120.)

[117] Vera, X., Lisper, B., and Xue, J. Data Caches in Multitasking Hard Real-Time Systems. In *Proceedings of the Real-Time Systems Symposium (RTSS)* (Cancun, Mexico, 2003). (Cited on pages 61 and 79.)

[118] Verma, M., and Marwedel, P. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. *IEEE TVLSI 14*, 8 (2006). (Cited on page 34.)

[119] Verma, M., and Marwedel, P. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors.* Springer, 2007. (Cited on pages 98 and 145.)

[120] Verma, M., Petzold, K., Wehmeyer, L., Falk, H., and Marwedel, P. Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach. In *Proceedings of the Workshop on Embedded System for Real-Time Multimedia (ESTIMedia)* (Jersey City, NJ, USA, 2005), pp. 115–120. (Cited on page 90.)

[121] Wehmeyer, L., Jain, M. K., Steinke, S., Marwedel, P., and Balakrishnan, M. Analysis of the influence of register file size on energy consumption, code size and execution time. *IEEE TCAD 20*, 11 (2001). (Cited on page 33.)

[122] Wehmeyer, L., and Marwedel, P. Influence of Onchip Scratchpad Memories on WCET prediction. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis (WCET)* (Catania, Sicily, Italy, jun 2004). (Cited on page 90.)

[123] Wehmeyer, L., and Marwedel, P. *Fast, Efficient and Predictable Memory Accesses - Optimization Algorithms for Memory Architecture Aware Compilation*. Springer-Verlag, 2006. (Cited on pages 89 and 98.)

[124] Weiser, M. The computer for the 21st Century. *SIGMOBILE Mobile Computing and Communications Review - Special 3* (1999), 3–11. (Cited on page 1.)

[125] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems 7* (2008), 36:1–36:53. (Cited on page 14.)

[126] Zemanek, H. Computer prehistory and history in central Europe. In *Proceedings of the National Computer Conference and Exposition* (New York, NY, USA, 1976), pp. 15–20. (Cited on page 1.)

[127] Zhao, W., Kulkarni, P., Whalley, D., Healy, C., Mueller, F., and Uh, G.-R. Tuning the WCET of Embedded Applications. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)* (Toronto, Canada, 2004), pp. 472–481. (Cited on page 12.)

[128] Zhao, W., Whalley, D., Healy, C., and Mueller, F. WCET Code Positioning. In *Proceedings of the Real-Time Systems Symposium (RTSS)* (Lisbon, Portugal, 2004), pp. 81–91. (Cited on pages 37 and 38.)

[129] Zitzler, E., Laumanns, M., and Thiele, L. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In

Proceedings of the Conference on Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN) (Athens, Greece, 2001), pp. 95–100. (Cited on page 39.)

[130] Zivojnović, V., Martinez, J., Schläger, C., and Meyr, H. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT)* (Dallas, TX, USA, 1994). (Cited on pages 65, 85 and 132.)

Memory-based Optimization Techniques for Real-Time Systems

Abstract:

Embedded/cyber-physical systems have become popular in a wide range of application scenarios. Such systems are called real-time systems if they underlie strict timing constraints. To verify if such systems can meet their deadlines, the knowledge of an upper bound for a program's execution time is mandatory. This upper bound is also called *worst-case execution time* (WCET) and is estimated by static timing analyzers.

Established optimizing compilers are not aware of the WCET as objective since they focus on the minimization of the *average-case execution time* (ACET). To overcome this obstacle, this thesis presents memory-based optimization techniques which focus on the reduction of the WCET of programs. All presented optimizations are integrated into the *WCET-aware C Compiler* (WCC) framework.

Since the memory interface of a system often turns out to be a bottleneck which limits the performance of a system, the presented optimizations are applied to different levels of the memory hierarchy of a system. Starting within a CPU core, the instruction fetch buffer is the most tightly coupled memory which tries to provide the next few instructions to be executed. Optimization techniques are presented improving the efficiency of this buffer w. r. t. the WCET of a system. Instruction caches placed between the CPU core and the main memory try to speed up accesses to the main memory by storing local copies in fast small cache memories. In order to improve the efficiency of this part of the memory hierarchy, a memory content selection approach is introduced which improves the WCET of a program by improving the cache performance.

Due to the fact that multi-task systems are employed in almost all domains, this thesis presents elaborate extensions to a compiler supporting the compilation and WCET-aware optimization of multi-task systems. These extensions exploited to develop a number of novel optimizations for systems running multiple tasks. As first optimization, a WCET-driven software-based cache partitioning demonstrates the effectiveness of considering the WCET for the optimization of a set of tasks. Furthermore, many embedded systems integrate so-called *scratchpad memories* (SPM) as tightly coupled memories. An optimization approach for SPM allocation in a multi-task scenario is proposed. Besides, a holistic view of memory architecture compilation considers a number of memory-based WCET optimizations and presents approaches for a combined application.

Existing compiler frameworks which are able to consider the WCET during optimization are limited to a particular hardware platform. In order to support multiple platforms, this thesis presents techniques to extend an existing WCET-aware compiler framework. Based on these extensions, a novel static cache locking optimization selects memory blocks which are statically locked into the instruction cache driven by WCET reductions.

Applying these optimizations, the WCET of real-time applications can be reduced by about 35% to 48%. These results underline the need for specialized WCET-driven optimization techniques integrated into a sophisticated compiler framework. Otherwise, immense optimization potential would remain unused resulting in oversized and thus costly embedded/cyber-physical systems.