Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DISSERTATION

Measurement-Based Timing Analysis of Applications written in ANSI-C

ausgeführt zum Zwecke der Erlangung des
akademischen Grades eines

Doktors der technischen Wissenschaften

unter der Leitung von

Univ.-Prof. Dr. Peter Puschner
Institut für Technische Informatik 182

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Bernhard Rieder
Matr. - Nr. 9325898
Forstgasse 16, 5500 Bischofshofen

Wien, im April 2009 ....

Measurement-Based Timing Analysis of Aplications written in ANSI-C

Since the development of the first electronic engine control systems the applications of electronic systems in cars have steadily increased. Currently there is a shift from mechanical safety-critical systems to networks of safety-critical embedded real-time systems which provide the potential for increased functionality and huge monetary savings. The increased functionality comes at the cost of increased complexity. The applications running in a modern luxury car come close to a total of 100 million lines of source code, distributed over networks of 70-100 microprocessor electronic control units (ECU) from different vendors.

To minimize errors in the design standardized architectures like AUTOSAR and communication buses like FlexRay™ or CAN are used to ensure the interoperability of modules. Model-checkers, verification and profiling tools are used to analyze the software of individual modules. The aspect of time is often underrated and many faults of control systems are a direct result of timing errors. During the last years a few timing analysis tools have emerged. Some of them use formal methods to calculate the execution-time of a task-based on a processor model, other tools use measurements to determine an estimate for the worst-case execution-time (WCET). Hybrid WCET analysis tools combine static analysis of the application source code, which makes this part of the analysis hardware independent, with execution-time measurements carried out on the target hardware to generate a hardware specific timing model of the analyzed application.

This work extends the hybrid timing analysis approach introduced during the MoDECS project to support loops, function calls and control-flow in logic AND and OR expressions which are required to measure the execution-time of industrial real-time applications. The revised hybrid WCET analysis approach comprises the following steps:

Static analysis is used on the ANSI-C source code to examine the program structure. C is commonly used in the implementation of control systems and the use of a high level language as input makes the analysis platform independent. During the static analysis functions are identified and either expanded like C++ inline functions or analyzed in a separate analysis run. Loops are also detected in this analysis step and checked for input data dependency. When required, which is when the number of iterations or the control-flow within the loop body depends on input data, the loop bound is determined using model checking. Last but not least additional control-flow paths which are generated by C short-circuiting of logic AND and OR expressions are analyzed and added to the control-flow graph.

Control Flow Graph (CFG) Partitioning is used to automatically split programs into smaller program segments (PS) which can be analyzed with reasonable effort.

Test Data Generation is used to generate test data to cover all paths within a program segment. Paths that are not covered by random test data are examined using model checking. Model checking is an expensive process but it can be used to generate test data to force the execution of a specific path within a program segment or to identify infeasible paths.

Execution Time Measurements are used on all paths within each program segment except the paths identified as infeasible during the test data generation. This produces a timing profile containing the worst-case execution-time for each program segment.

Worst Case Execution Time (WCET) Calculation uses the structural information gained during the static analysis to combine the execution-times of individual program segments into a WCET bound for the whole analyzed application.

Messbasierte Zeitanalyse von ANSI-C konformen Applikationen

Seit der Entwicklung der ersten elektronischen Motorsteuerungssysteme hat die Elektronik in immer schnelleren Schritten Einzug ins Automobil gehalten und ist derzeit dabei, sicherheitskritische mechanische Systeme abzulösen. Anstatt mechanischer Systeme werden jetzt verstärkt Netzwerke aus eingebetteten elektronischen Steuergeräten eingesetzt, welche erhöhte Funktionalität bei verringerten Kosten versprechen. Die erweiterte Funktionalität wird mit zunehmender Komplexität der Systeme erkauft. So beinhaltet die Elektronik eines modernen Luxusmodells heute in etwa 100 Millionen Zeilen an Programmquelltext, verteilt auf Netzwerke aus 70-100 software-gesteuerten Steuergeräten verschiedener Hersteller.

Um Fehler im Design durch mangelnde Interoperabilität zu vermeiden, setzt man auf verschiedene standardisierte Architekturen wie AUTOSAR und Netzwerke wie FlexRay™ und CAN. Die Software in den Steuergeräten wird mittels Modellprüfern, Software-Verifikations- und Analysetools geprüft, wobei jedoch oft die Analyse des Zeitverhaltens vernachlässigt wird. In letzter Zeit wurden einige Werkzeuge zur Überprüfung des Laufzeitverhaltens von Software entwickelt, wobei ein Teil auf formale Methoden und abstrakte Prozessormodelle und der andere Teil auf Laufzeitmessungen an der Zielhardware setzt, um die maximale Ausführungszeit von Programmen (WCET) zu ermitteln. Hybride WCET Analyse verbindet plattformunabhängige statische Analyse des Anwendungs-Quellcodes mit Laufzeitmessungen, die die Ausführungszeit auf der Zielplattform messen und dadurch ein hardwareabhängiges Laufzeitprofil der Applikation erzeugen.

Diese Arbeit erweitert die hybride Zeitanalyse, die im Zuge des MoDECS Projektes entwickelt wurde, um Schleifen, Funktionsaufrufe und Kontrollflussentscheidungen in logischen UND und ODER Ausdrücken analysieren zu können, was für die Analyse industrieller Anwendungen unerlässlich ist. Die erweiterte WCET-Analyse besteht aus folgenden Schritten:

Statische Analyse des ANSI-C Quellcodes, um dessen Struktur zu ermitteln. C wird oft bei der Programmierung von Steuergeräten verwendet und macht als Ausgangsbasis für die Analyse diese plattformunabhängig. Während der Analyse werden Funktionsaufrufe erkannt und die aufgerufenen Funktionen ähnlich wie C++ inline-Funktionen expandiert oder in einer getrennten Analyse untersucht. Schleifen werden in der statischen Analyse erkannt und auf ihre Eingabedatenabhängigkeit untersucht. Bei eingabeabhängigen Iterationsbedingungen oder eingabeabhängigem Kontrollfluss in der Schleife wird in diesem Schritt auch deren maximale Iterationszahl mittels Modellprüfung bestimmt. Letztendlich werden bei der statischen Analyse noch logische UND- und ODER-Ausdrücke untersucht und im Kontrollflussgraph eingetragen.

Die *Partitionierung des Kontrollflussgraph (CFG)* teilt den CFG automatisch in kleinere Programmsegmente (PS) auf, die mit vertretbarem Aufwand analysiert werden können.

Die *Testdatengenerierung* dient dazu, Testdaten für alle möglichen Ausführungspfade in einem PS zu generieren. Modellprüfung wird benutzt, um Testdaten für Pfade zu finden, die durch die erzeugten Zufallsdaten nicht abgedeckt werden konnten und um Pfade die aufgrund der Codesemantik unausführbar sind als unausführbar zu identifizieen.

Laufzeitmessungen dienen zur Ermittlung der Ausführungszeit aller tatsächlich ausführbaren Pfade eines Programmsegments. Die Laufzeitmessungen erzeugen ein Ausführungszeitprofil jedes Programmsegmentes, das unter anderem auch die WCET des PS beinhaltet.

Die *Berechnung eines Wertes für die maximalen Ausführungszeit (WCET)* durch Kombination der einzelnen Laufzeitprofile wird unter Berücksichtigung der Strukturinformation aus der statischen Analyse durchgeführt, um die WCET der analysierten Applikation zu erhalten.

Acknowledgements

First of all, I would like to thank my supervisor and professor Univ. Prof. Dr. Peter Puschner who encouraged me with his helpful advice and valuable support. He supported my work with valuable suggestions and without his help this thesis would not be finished yet. I would also thank my secondary advisor Univ. Prof. Dr. Jens Knoop for his willingness to evaluate this thesis on short notice and for his suggestions how to improve it. I am also grateful to the head of the department of the Institut für Technische Informatik, Real-Time Systems Group, O. Univ. Prof. Dr. Hermann Kopetz who made it possible to me to work in a highly motivated and motivating environment.

I would also like to thank all my current and former colleagues of the institute for interesting discussions after lectures in the institute or over the occasional cup of coffee in the kitchen. I also thank the administrative staff of the institute for shielding me from administrative overhead as good as possible.

Especially I would like to mention Dr. Ingomar Wenzel with whom I worked together on the *MoDECS* project, which provided a solid foundation for my research. Dr. Raimund Kirner was also an interesting partner for discussions and supported me with valuable advice and ideas. I would also like to thank Dr. Klaus Steinhammer who designed the hardware for the measurement device used for the HCS12 architecture in the *MoDECS* project who was a big moral uplifting in the final stages of this work.

As a personal note I thank my family and my friends for their support during the writing of this thesis.

The research for this thesis has been conducted during my employment as a research assistant at the Institut für Technische Informatik, Real-Time Systems Group within the Vienna University of Technology which was financed by the FIT-IT project "MoDECS-d – Model-based Development of distributed Embedded Control Systems" under project number 807144/7855 and the FFG project "ATDGEN – Automatic Test Data Generation for WCET Measurements" under project number 812653/1729-GLE/BLC.

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Contribution . . . 3

1.3 Document Structure . . . 4

2 Basic Concepts **5**

2.1 The C Language Family . . . 5

2.1.1 Drawbacks of ANSI/ISO-C for Embedded Programming . . . 6

2.1.2 Language Features Support in this Thesis . . . 7

2.2 Code Transformation and Representation . . . 9

2.2.1 Compilation . . . 10

2.2.2 Lexical Analysis, Parse Tree and Abstract Syntax Tree . . . 10

2.2.3 Basic Block . . . 11

2.2.4 Control Flow Graph . . . 12

2.2.5 Path . . . 12

2.2.6 Dominator and Postdominator . . . 13

2.2.7 Program Segment . . . 13

2.3 Target Platform Complexity . . . 13

2.3.1 Influences of the Operating System . . . 14

2.3.2 Influences of the Compiler . . . 16

2.3.3 Influence of the Target Hardware . . . 19

2.3.4 Hardware Optimization Techniques . . . 19

2.4 Software Complexity . . . 29

2.4.1 Software Metrics . . . 30

2.4.2 Coding Guidelines . . . 31

2.5 Summary . . . 32

3 Execution Time Analysis **33**

3.1 Why Timing Analysis . . . 33

3.2 Worst-Case Execution Time (*WCET*) . . . 35

3.3 Timing analysis . . . 36
3.3.1 Anatomy of a WCET Analysis Framework . . . 37
3.3.2 Syntax Analysis and CFG extraction . . . 37
3.3.3 Static Analysis . . . 38
3.3.4 Dynamic Timing Analysis . . . 41
3.3.5 WCET Calculation . . . 46
3.4 The Measurement-Based or Hybrid Timing Analysis Approach . . . 49
3.4.1 Assumptions and Prerequisites . . . 50
3.4.2 Basic Idea - Guiding Measurements by Static Analysis . . . 51
3.4.3 Partitioning or Segmentation of the CFG . . . 52
3.4.4 Test data generation . . . 54
3.4.5 Function Calls . . . 55
3.4.6 Simplified Data Flow Analysis . . . 55
3.4.7 Loops . . . 56
3.4.8 WCET Calculation Step . . . 56
3.4.9 Differences to MoDECS V2 . . . 56
3.5 Summary . . . 57

4 Execution-Time Analysis Framework **59**
4.1 Development Environment . . . 59
4.2 Example Application . . . 59
4.3 Input Parsing . . . 61
4.4 Code Generation . . . 61
4.5 CFG Generation . . . 61
4.6 Expression Paths . . . 64
4.7 Const Analysis . . . 66
4.8 Loop Analysis . . . 67
4.8.1 Loop Categorization . . . 67
4.8.2 Loop Bound Analysis . . . 69
4.9 Counting Paths Between Two Nodes . . . 70
4.10 Dominator and Postdominator Tree . . . 73
4.11 Segmentation . . . 73
4.12 Decision Tree (dtree) . . . 76
4.13 Test Data Generation . . . 78
4.13.1 Reused Test Data . . . 78
4.13.2 Random Test Data . . . 79
4.13.3 Model Checking . . . 80

4.14 Managing Test Data 80
4.15 Measurements 81
4.15.1 Generating Target and Module Code 82
4.16 Analysis Tool Usage and Output 84
4.16.1 Parameter File 84
4.16.2 Command Line Arguments 85
4.16.3 Tool Output 86
4.17 Summary 87

5 Experiments **89**
5.1 Test Setup 89
5.1.1 Basic Block and Path Counts Explained 89
5.1.2 Selection of Case Studies 91
5.1.3 Test Hardware and Development Software 95
5.1.4 Target Software Layout and Host-Target Communication . . . 97
5.2 Description and Goals of Tests Scenarios 100
5.3 Loops 101
5.3.1 General Loops 101
5.3.2 General vs. Specialized Loop Handling 101
5.3.3 Reduced Overestimation for 1:n-Loops 104
5.4 Function Inlining vs. Black-Boxing 106
5.5 Control Flow in Expressions 109
5.6 Results 111
5.6.1 Loops 111
5.6.2 Function Inlining vs. Black-Boxing 112
5.6.3 Control Flow in Expressions 113
5.7 Summary 114

6 Related Work **115**
6.1 WCET Analysis 115
6.1.1 Static WCET Analysis 116
6.1.2 Measurement Based and Hybrid WCET Analysis 121
6.1.3 Overview of Current WCET Analysis Tools 126
6.2 Improving the WCET 126
6.2.1 WCET-Oriented Programming 126
6.2.2 Single-Path Conversion 129
6.3 Model Checking 130
6.4 Program Analysis 133

6.5 Cache Analysis 134
6.6 Timing Anomalies 135
6.7 Other Publications related to this Work 138
6.8 Conclusion 138

7 Conclusion and Outlook 141
7.1 Summary of Measurement-Based WCET Analysis 141
7.2 Lessions Learned 142
7.2.1 Gained Experiences 143
7.3 Applications of Hybrid WCET Analysis 143
7.4 Future Work 144

Bibliography 147

Index 165

A List of Abbrevations 169

B Acknowledgements 171

List of Figures

2.1 Code generation for Logical AND and OR Short-Circuiting 9
2.2 The Compiler Abstraction 10
2.3 Illustration of Basic Concepts 11
2.4 The S-task model 16
2.5 DRAM vs. SRAM Cell and Processor Die with SRAM Cache 21
2.6 AMD Opteron® Memory Hierarchy 22
2.7 Simple MIPS Pipeline 23
2.8 Pipelined Execution of Instructions 24
2.9 2-bit Branch Prediction 26
2.10 Normal Opcodes and Predicated Execution Opcodes 27
2.11 Variable Execution Times on the Intel® x86 Processor Family 28
2.12 A Timing Anomalie Example 29

3.1 Timing Requirements 33
3.2 Response Time 34
3.3 Execution Time Distribution of a Task 36
3.4 Overview of different WCET calculation methods 47
3.5 Basic Hybrid WCET Analysis Approach 51
3.6 Concept of CFG Partitioning 53
3.7 Test Data Generation 54

4.1 Running Example for Chapter 4 60
4.2 Basic CFG building algorithm 63
4.3 Shortcut Paths in Logical Expressions 65
4.4 Instrumentation of Logic Expressions 66
4.5 Const Analysis 68
4.6 Model Checking for Loop Bound Calculation 70
4.7 CBMC Output for Loop Bound Checking 71
4.8 Path Counting Algorithm 72
4.9 DOM and PDOM tree for example.c 74

4.10 Segmentation Algorithm . . . 75
4.11 DTree for the Example Application . . . 77
4.12 Random Value Distribution for Test Data Generation . . . 79
4.13 Model Checking to find a Path inside the Loop . . . 81
4.14 Example xml Data File . . . 82
4.15 Generated Loadable Module . . . 83
4.16 Parameter File . . . 85
4.17 WCET analysis Tool Help Text . . . 86
4.18 WCET analysis Tool Version Information . . . 87
4.19 Tool Output . . . 88

5.1 Code Metrics Example . . . 90
5.2 Code generation options for industrial case studies . . . 92
5.3 Olimex® LPC-H2138 Development Board and Programming Device . . . 95
5.4 Measurement Instructions . . . 98
5.5 Linker Command File for *Caller* . . . 99
5.6 1:n-Loop with ET for Different Iterations . . . 104
5.7 A complex 1:n Loop in the Binary Search Test Case . . . 105
5.8 Inlining vs. Black-Boxing . . . 106
5.9 Handling of Expression Paths in V2 and V3 . . . 110
5.10 Listing of *fibcall* Case Study . . . 113

6.1 Comparison of Traditional and WCET-Oriented Programming . . . 127
6.2 Single Path Programming . . . 130
6.3 Extraction of Bit-Vector Equations with CBMC . . . 132
6.4 Timing Anomalies Examples . . . 136

List of Tables

5.1 Description of Test Cases . . . 94
5.2 Analyzeable Test Cases in V2 and V3 . . . 102
5.3 Generic and Spezialized Loop Handling . . . 103
5.4 Comparison between Inlining and Black-Boxing . . . 108
5.5 Measurement Results for the Expression Path Example . . . 111
5.6 Effects of Control Flow in Expressions in Industrial Case Studies . . . 111

6.1 Overview of WCET Analysis Tools . . . 128
6.2 Execution time of traditional and WCET-oriented Algorithms . . . 129

Chapter 1

Introduction

The software systems of modern cars have reached a sizes of 100.000.000 lines of code and more [Cha09], implementing or controlling more than 2000 functions in premium cars and consuming 50 to 70 percent of the development costs of software/hardware subsystems of cars [LLS+07]. The software is distributed among multiple networks of up to 70 Electronic Control Units (*ECU*s) from different vendors which have to work together as a distributed real-time system. Today distributed real-time systems in cars are primarily used for non safety-critical functions, mainly because these systems are still young (the first car series to use the FlexRay® communication network was the BMW X5 in 2006) and the industry needs to evaluate their dependability. In aviation the Time Triggered Protocol (TTP®) is the first real-time communication protocol and it has been used in the cabin pressure control system of Airbus' superjumbo A380 since 2006.

It can be expected that distributed real-time systems will be used for safety-critical applications in the near future and are soon to replace mechanical and hydraulic subsystems. Current systems are time-triggered, which means that events are periodically transmitted over the communication channel rather than only when an event occurs. This ensures that the communication channel can handle multiple simultaneous events without event loss. The communication schedule, which specifies which node sends a data frame in a specific communication slot during a communication cycle is specified during system design, along with the definition of the individual data items within the frame. This approach provides a well-defined interface between communication nodes and facilitates composability and component-based design.

In well designed architectures not only the communication but also the task scheduling in each node is time-triggered and synchronous to communication cycles. This allows a safe calculation of the maximum delay between an event and the response caused by the controlling system. When an event occurs it is precisely known which node performs which sub-tasks of the event processing at which time, making the system fully predictable and easily to diagnose. The cycle time is shared between all running tasks. To ensure the execution of each task within a cycle, the sum of the Worst Case Execution Times (*WCET*) of all tasks has to be less than

the cycle time. On small real-time systems, using only a single computation node, cycle-based approaches are also widely used since it is easier to ensure response times for time-triggered architectures than for event-triggered architectures. However, both approaches require the WCET to be known to ensure a maximum response time of the system. In time-triggered architectures the WCET is also required to

If the exact WCET is not known, which applies for most cases, an overestimation of the WCET, which is referred to as safe upper bound, can be used for system design. A popular approach to gain a "safe" upper bound for the execution-time is to perform measurements of the execution-time using "typical" test data or synthetic test data which is supposed to enforce an execution path through the program which generates the WCET. The result, which consists of the maximum of the measured execution-times plus a safety margin, is used as a "safe" upper bound for the WCET. Code generator tools like the Matlab®/TargetLink® toolchain offer to perform this task automatically. However, there are no guarantees that the WCET is equal or less than the worst observed execution-time plus the safety margin.

There exist many tools which allow the verification of the value domain of software, which means that the correct value of the result is checked. The time domain, which considers the timeliness of the result, is currently neglected. At the moment there exist only few tools for timing analysis. They are listed in Chapter 6. The majority of these tools are feasibility studies and research prototypes. Only a small fraction of the presented tools are commercially available, but they also have their shortcomings. To make things worse, some tools which operate on a low abstraction level like object code, can only be used for a specific target architecture. Concluding it can be said that the current state of (integrated) timing analysis tools is more than unsatisfactory.

1.1 Motivation

As the software portion in safety-critical areas of applications such as automotive or avionic applications is steadily increasing, the lack of applicable timing analysis tools, which can be embedded within an existing application development framework for (distributed) real-time applications, becomes an increasing safety issue and a major showstopper for the integration of embedded real-time systems in safety-critical systems.

The goal of this work is to develop a timing-analysis method and a prototype tool which is easy to use, operates on multiple target platforms and can be integrated into existing development processes to find a good approximation the WCET. Easy usage means that no extensive preparation like annotations or manual test data generation shall be necessary to perform timing analysis. A basic knowledge of the matter is however necessary to get reliable data. Platform independence is achieved by using the source code as basis for the analysis. Only a thin hardware dependent interface is required to perform the run-time measurements as well as basic knowledge of the bit size of the basic data types and the endianess of the target architecture. Easy

integration can be achieved by a well defined interface of the execution-time analysis tool to other applications.

The developed execution-time analysis method is an extension to the measurement-based worst-case execution-time analysis (*MBTA*) method developed during the MoDECs Project [WRKP05] to support loops, function calls and control-flow paths within boolean AND (&&) and OR (||) expressions and the conditional expression operator (? :). The developed analysis tool is not intended as a general WCET analysis tool but to test the developed analysis method.

Hopefully the results of this work will help to improve the timing analysis process for embedded real-time applications.

1.2 Contribution

The major contributions of the measurement-based worst-case execution-time analysis method presented in this thesis are:

1. **Avoidance of explicit hardware modelling and target platform independence*** Static execution-time analysis methods require a detailed hardware model of the processor to calculate the execution-time of a series of instructions. Dynamic methods execute the code on the target platform and obtain the execution-time using measurements. In cases where no detailed information about the target hardware is available static execution-time analysis cannot be performed while dynamic execution-time analysis is still possible.

2. **Automatic control-flow generation for shortcut boolean expressions** C defines a feature called short-circuit code, where boolean expressions are only evaluated as far as required. This speeds up calculation but also increases the number of possible execution paths. WCET analysis has to handle these expressions accordingly.

3. **Parameterizable control-flow graph (*CFG*) partitioning*** The complexity of the analysis can be reduced by splitting the analyzed program into program segments (*PS*) as shown in Chapter 4. Increasing the number of PS reduces the number of required measurements but may lead to overestimations of the execution-time.

4. **Automated test data generation using model checking*** Model checking can be used to generate the test data for all feasible paths or identify infeasible paths within a given program segment. The coverage of all feasible paths is a cornerstone of the proposed execution-time analysis method.

*This has been contributed together with Ingomar Wenzel during the MoDECS project [WRKP05].

5. **Detection of loop bounds using model checking** In order to perform measurements of loops the loop bound has to be known. The proposed solution is to use model checking to find loop bounds [RPW08].

6. **Automatic classification and measurement of loops** Different loops require individual approaches for measurement, depending on the control-flow structure of their loop body and the loop iterator. The presented prototype can automatically detect different kinds of loops and perform the analysis accordingly.

7. **Inlining of function calls to reduce control-flow of functions** WCET estimates of functions may be shorter when the analysis considers the actual context of the function call, and not just the overall maximum execution time of the function. The presented prototype achieves this by inlining function code during the path-analysis phase of the WCET analysis on demand.

1.3 Document Structure

This thesis is structured as follows:

Chapter 2 discusses basic features of the C programming language, basics of compiler construction and how the complexity of the analysis is influenced by the operating system, the compiler and the target hardware. Then, Chapter 3 discusses the basics and different approaches of execution-time analysis and proposes a novel measurement-based execution-time analysis (*MBTA*) method. Following, Chapter 4 explains how the proposed MBTA method is realized within the prototype application. In Chapter 5 the individual experiments conducted with the WCET analysis prototype and why they were chosen are described, followed by a short discussion of the results. Chapter 6 explains similar work and their relation to this thesis. Finally, Chapter 7 concludes this work and summarizes the findings from the MTBA method, the prototype implementation and the conducted experiments. The chapter ends with an outlook which shows ideas for further reseach.

Chapter 2

Basic Concepts

This chapter gives a closer look at the different C standards and explains terms and concepts frequently used in this thesis. In this chapter it is also shown what makes timing analysis difficult and how the target platform, the operating system and the source code influence the complexity of the analysis. The influence of coding guidelines for automotive and avionic applications on execution-time analysis is also examined in this chapter.

2.1 The C Language Family

The C Programming Language has been continuously evolving since its early beginnings in 1969. The following paragraphs give a short overview about the different versions of C and explains which features of the different versions are supported in the timing analysis framework prototype presented in this thesis.

K&R C

The first edition of "The C Programming Language" [KR78] was published in 1978 by Brian Kernighan and Dennis Ritchie. The name of the language "C" was derived to indicate that C was designed to be a successor of the "B" programming language. Even after C was standardized by ANSI (in 1989) and ISO (in 1990) K&R C still was considered the de-facto standard since it was supported with all its features by the majority of the compilers. The style of function definitions used by this early version of C is not supported by the timing analysis framework.

C89 and C90

After a six year lasting standardization process the C standard was ratified by the American National Standards Institute as ANSI X3.159-1989 "Programming Language C" [C89]. This version, commonly referred to as *ANSI-C* or *C89* was adopted

by the International Organization for Standardization (ISO) as ISO/IEC 9899:1990 [C90], which is referred to as *C90* with only minor modifications. Since this standard is currently used for the majority of embedded applications the execution-time analysis tool presented in this thesis is currently based on this standard.

C99

The ISO/IEC 9899:1999 Standard [C99] which adds C++-style comments, inline functions and variadic macros (macros with a variable number of arguments) was adopted by ANSI in early 2000 and is referred to as *C99*. As the extensions are usually not used for embedded and real-time programming, the language extensions provided by C99 are not supported by the timing analysis framework presented in this thesis.

C++

C++ which was standardized in 1998 and revised 2003 is an object-oriented extension to C. The name "C++" is an allusion to the ++-operator of C and indicates that C++ is an enhanced successor of C. Due to its heavy use of polymorphism and dynamic memory C++ is not suited for embedded and real-time applications and therefore not considered in this thesis.

2.1.1 Drawbacks of ANSI/ISO-C for Embedded Programming

The C standard leaves important aspects undefined and includes features that are not suited for embedded or real-time programming. Some important problems of standard compliant *ANSI/ISO-C* code are listed below. A detailed list of undefined and platform dependent or implementation specific behavior can be found in the Annex J of the ISO/IEC C99 standard [C99].

Undefined Size of Data Types

The C definitions of integer data types are very vague. For instance a char is composed of any number of bits as long as the number of bits is larger than or equal to eight. Also the binary layout (big endian, little endian, or even gray) is unspecified. This restricts platform independence and portability and requires deep knowledge about the target platform. As a result the operators working on integers are not be well-defined, too. There are some libraries and target specific headers trying to overcome this limitation and defining datatype like SINT8, UINT8, SINT16, ... etc, but each vendor has his own concepts and there is no standard way to write platform independent code.

Undefined Handling of Errors

Like the integer types some error conditions are undefined. What happens on divisions by zero or on illegal memory access operations? This behavior is mostly operating system dependant. Therefore not only precise knowledge about the target hardware but also detailed information about the target operating system is necessary in order to describe the complete semantics of C programs.

Weak Type Checking

The type checking in C is weak, that means any type can be interpreted as another type using a simple type cast. While this makes it very convenient for the programmer to manipulate binary data efficiently or to access single bits of hardware registers on the target it makes verification difficult. Sometimes a type cast is performed without the programmer being aware of it. If a type with a large value range is converted to a type with a smaller value range this can even lead to information loss (however a compiler should issue a warning when this happens).

Dynamic Memory

Dynamic memory is a feature that is often used in C but which is very problematic for embedded systems. First of all it can lead to memory fragmentation and second, the time consumption of memory allocation and deallocation cannot be predicted. This does not imply that dynamic memory cannot be used in embedded systems at all, but all memory (de-)allocation operations have to be performed in an non real-time initialization task. However, this makes analysis and verification very difficult and dependent on the memory (de)allocation stategy used by the operating system.

2.1.2 Language Features Support in this Thesis

A goal of this thesis is to be as target independent as possible. Therefore the presented Runtime Analysis Toolkit works on source code. As mentioned before, additional knowledge about the target is necessary.

Type Qualifiers and Storage Class Modifiers

The framework recognizes *const* declarations and uses the declared variables as constants in the simplified data flow analysis. The *volatile* declaration is ignored, since the proposed approach uses model checking and there is no feasible way to represent the concept of a volatile variable in a model checker simulation. When the value is used only in the data flow this is no problem. When a single control-flow decision depends on *volatile* this is also no problem because the model checker will examine all possible paths. However, a problem occurs when a path depends on

changing values of volatile variables. A solution to this problem is to create multiple variables, one for each use of the volatile variable in the control-flow but this was not implemented due to its little relevance.

Storage class modifiers, especially *typedefs* are supported. *Extern* declarations are supported as long as the declared variable exists within one of the given input files. *Static* global variables are supported and only recognized in the same source file. *Static* function variables are prefixed with a unique ID and moved to global scope. This is necessary since the control-flow decisions in an application can depend on the state of the application which is often stored in static function variables. *Auto* and *register* have no impact on the analysis and are ignored by the analysis. However, they are of course considered by the code generation for the target.

Pointers and Dynamic Memory

Dynamic memory is not supported since it is forbidden by the MISRA coding guidelines [MIS98, Rule 118], which are described briefly in section 2.4.2. Pointers are supported as long as they are not used as static variables or function parameters for the examined function, which means they have to be initialized to point to a known variable or constant before they are used. A typical application for pointers which is supported by the framework is to iterate over an array using a pointer to access individual data elements.

Floating Point Operations and Bit Operations

Applications may contain floating point variables and operations as long as they do not influence the control-flow (section 2.2.3). Bit operations like AND, OR and XOR are fully supported.

Structs and Unions

Structs are supported. Unions are disencouraged by the MISRA coding guidelines [MIS96, MIS98, MIS04] discussed in section 2.4.2 and therefore not supported.

Logical AND, OR and Conditional Expressions

Logical AND (&&) and OR (||) operators are not necessarily completely evaluated in C (Short-Circuit Code [ASU86, ALSU06]). If the first operand evaluates to 0 (&&) or to 1 (||) the right side of the operator is skipped. This behavior, which is called expression short-circuiting, is defined in [C99, Chapters 6.5.13. and 6.5.14]. Figure 2.1 shows how logical AND and OR are typically translated to object code. This creates a conditional execution path that has to be considered by the analysis. Since many applications rely on the feature of C shortcut evaluation it is supported

```
x == a && b;
```

(a) AND expression in C

$$
\begin{array}{ll}
 & if\ a = 0\ goto\ \mathrm{L1} \\
 & if\ b = 0\ goto\ \mathrm{L1} \\
 & x \leftarrow true \\
 & goto\ \mathrm{L2} \\
\mathrm{L1}: & \\
 & x \leftarrow false \\
\mathrm{L2}: & \\
 & \ldots
\end{array}
$$

(b) Generated Code for AND

```
x == a || b;
```

(c) OR expression in C

$$
\begin{array}{ll}
 & if\ a \neq 0\ goto\ \mathrm{L1} \\
 & if\ b \neq 0\ goto\ \mathrm{L1} \\
 & x \leftarrow false \\
 & goto\ \mathrm{L2} \\
\mathrm{L1}: & \\
 & x \leftarrow true \\
\mathrm{L2}: & \\
 & \ldots
\end{array}
$$

(d) Generated Code for OR

Figure 2.1: Code generation for Logical AND and OR Short-Circuiting

by the presented execution-time analysis framework. Similarily the conditional expression operator (?:) introduces control-flow paths. However, when writing "?:" it is more obvious that additional execution paths are created.

Loops

In general loops are supported. Details about which types of loops are supported can be found in chapter 3.4.7. Nested loops are not supported by the WCET analysis prototype.

Library Calls and Functions

Library calls are supported as long as the library is available in source code. Not only may the execution-time of library functions vary but the library function has to be traceable by the model checker. The same applies to user defined functions. Only a single function call per expression is supported. Therefore constructs like `outer(middle(inner(x)));` or `f1(x)+f2(x);` are not supported. Additionally function calls are not supported in logical expressions with espression short-circuiting.

2.2 Code Transformation and Representation

The execution-time analysis framework presented in this thesis uses methods for program representation, analysis and transformation which are well known in the context of compiler construction. These basic concepts of compiler construction techniques are briefly presented in this section. Some of the following definitions can also be found in similar form in [Wen06]. Since the internal data representation of the presented execution-time analysis prototype differs from the prototype created during the *MoDECS* project some of the definitions have been altered to fit this version.

2.2.1 Compilation

Definition 1 *Compiler*
A Compiler C is a program that reads a program written in a source Language S - the source language - and translates it into an equivalent program in another language T - the target language. As an important part of this translation process, the compiler reports to its user the presence of (syntactical) errors in the source program (figure 2.2) [ASU86, ALSU06].

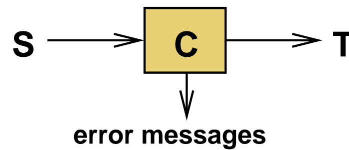


Figure 2.2: The Compiler Abstraction

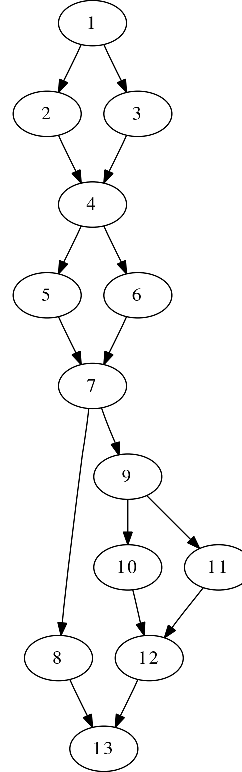
The compiler has to ensure that the program semantic is not changed by the transformation. Typically a compiler generates executable code for a specific platform which consists of operating system and hardware. Alternatively another representation can be generated which cannot directly be executed on the target platform but uses a platform dependent interpreter (i.e. java bytecode). Typically compilation steps include program analysis (*lexical analysis*, *syntax analysis*, *semantic analysis*), and code generation (*intermediate code generation*, *code optimization*, *code generation*). A compiler usually consists of three independent operating and exchangeable parts. The frontend handles a specific input language and translates it to an internal representation. The middleend performs platform independent analyses (i.e. data-flow analysis, control-flow a., pointer and alias a., and more) and the backend optimizes the program for a special target architecture and generates an object file that can be executed on the target hardware.

2.2.2 Lexical Analysis, Parse Tree and Abstract Syntax Tree

A parse tree is a data structure that is typically generated while parsing a program, using the output of the lexical analysis (*tokens*) and allowing a literal identical reconstruction of the program (except for whitespaces) [ASU86, ALSU06]. An Abstract Syntax Tree (*AST*) is an abstraction of a parse tree which provides semantically equivalent information like a parse tree but does not allow the exact reconstruction of the program source, especially literals like variable names may be replaced by more suiting representations. As recreation of the source code is a requirement for the introduced execution-time analysis tool, abstract syntax trees are used in this thesis and all code modifications take place in this representation.

```
 1 int calc (int a,
 2           int b, int c)
 3 {
 4   int rv;
 5   rv = 0;           // 1
 6   if (a > 0) {
 7     rv += a;        // 2
 8   } else {
 9     rv -= a;        // 3
10   }
11   if (b > 0) {      // 4
12     rv += b;        // 5
13   } else {b
14     rv -= b;        // 6
15   }
16   if (c == 0) {     // 7
17     rv = 1;         // 8
18   } else {
19     if (c > 0) {    // 9
20       rv *= c;      // 10
21     } else {
22       rv *= -c;     // 11
23     }
24
25     rv = rv+1;      // 12
26   }
27   return rv;        // 13
28 }
```

(a) Listing



(b) Control Flow Graph (CFG)

Figure 2.3: Illustration of Basic Concepts

2.2.3 Basic Block

Definition 2 *Basic Block*
A Basic Block BB *is a linear sequence of expression statements with the following properties:*
(a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
(b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

This definition can be found in similar form in [ASU86, ALSU06]. In a basic block there are no branches in the *control-flow* (see below). An example for basic blocks can be seen in figure 2.3. Each node in subfigure b corresponds to a basic block. The beginning of each basic block is marked with a comment that displays the number of the basic block in subfigure a.

2.2.4 Control Flow Graph

Definition 3 *Control Flow*
The control flow describes the order in which the individual statements of a program are executed or evaluated.
Control Flow Statements are statements that decide which control-flow out of a set of alternatives should be followed.

Preemptive multitasking, interrupts and signals also alter the control-flow but are out of scope of this thesis since only simple tasks (see section 2.3.1) are considered.

Definition 4 *Control-Flow Graph* CFG
A control-flow graph is a quadruple $G = (N, E, s, t)$ consisting of a set of nodes N, representing the BBs of a program, a set of directed edges E, representing the control-flow between the BBs, a dedicated entry node $s \in N$ and an exit node $t \in N$. The set of predecessor and successor nodes for a node n are defined as $pred(n) = \{(a, n) \in E \mid \forall a \in N\}$ and $succ(n) = \{(n, b) \in E \mid \forall b \in N\}$. The entry node s has no predecessors, i.e. $pred(s) = \emptyset$, and the exit node t does not have any successors, i.e. $succ(t) = \emptyset$. For all $e_i = (n_1, n_2) \in E$ holds $n_1, n_2 \in N$. Each node $n \in N$ is reachable from the entry node s and the exit node t is reachable from each node $n \in N$ by following a finite number of edges $e \in E$.

The example presented in figure 2.3(b) shows a CFG generated from the listing in figure 2.3(a).

2.2.5 Path

Definition 5 *Paths*
A path π from basic block n_0 to basic block n_n is a sequence of nodes $\pi = (n_0, n_1, \ldots, n_n)$ in a given CFG $G = (N, E, s, t)$ with $(n_i, n_{i+1}) \in E$. The length of the path π is defined as $|\pi| = n$.
Paths starting from $n_0 = s$ to $n_n = t$ are called execution paths and represent the control-flow for a single execution of a program. Π denotes the set of all possible execution paths within G. For each $n \in N$ there exists at least one execution path $\pi_n \in \Pi$ where $n \in \pi_n$.

The possible paths through the example CFG shown in figure 2.3 are given by $\Pi = \{(1, 2, 4, 5, 7, 8, 13), (1, 3, 4, 5, 7, 8, 13), \ldots, (1, 3, 4, 6, 7, 9, 11, 12, 13)\}$ with $|\Pi| = 12$. The number of possible execution paths $|\Pi|$ increases exponentially with the size [LME99] or number of branches in the program.

Loops and goto statements can cause basic blocks to be executed more than a single time and generate cyclic paths π_c within a CFG G. Therefore we define cyclic paths as follows:

Definition 6 *Acyclic and Cyclic Paths*
An acyclic path π_{ac} *is a path where* $i \neq j \rightarrow n_i \neq n_j$ *holds in CFG G.*
A cycle $\theta = (n_i, \ldots, n_{j-1})$ *is a sub-sequence of* π *where* $i \neq j \wedge n_i = n_j$. $\Theta(\pi)$ *defines the set of all cycles within a given path* π. *A path containing cycles, i.e.* $|\Theta(\pi)| \neq 0$, *is called a cyclic path* π_c.
The edge $(n_{j-1}, n_j) \in E$ *which links to a previously visited node and therefore generates the cycle is denoted as a backedge.*

2.2.6 Dominator and Postdominator

Dominators (DOM) and Postdominators ($PDOM$) describe relations of the control-flow between basic blocks.

Definition 7 *Dominator*
A basic block n *dominates a basic block* o *when every path that reaches* o *has to pass through* n *which is denoted as* $n = dom(o)$. *The entry block* s *of a CFG* G *dominates all blocks within* G.

Definition 8 *Postdominator*
A basic block n *postdominates a basic block* m *when every path from* m *to the exit node* t *has to pass through* n. *This can be expressed as* $n = pdom(m)$. *The exit block* t *of a CFG* G *postdominates all blocks within* G.

2.2.7 Program Segment

Definition 9 *Program Segment (*PS*)*
A program segment PS is a subset of the CFG G. Each $PS_i = (N_i, E_i, S_i, T_i)$ *is a quadruple of basic blocks* $N_i \in N$, *directed edges* $E_i \in E$, *entry nodes* $S_i \in N$ *and exit nodes* $T_i \in N$ *where for all* $e = (n_1, n_2) \in E_i$ *holds* $(n_1, n_2 \in N_i) \vee (n_1 \in T_i \wedge n_2 \in N)$.

This definition of a PS allows multiple entry and exit nodes with the exit edges being part of the PS. In the execution-time analysis framework presented in this thesis only PS with $|E_i| = 1|$ and $|S_i| = 1|$ are used. It is possible to split a whole program down into arbitrary PS, a process which is referred to as *segmentation*. The proposed approach uses non-overlapping PS; therefore $\forall i, j, i \neq j : n \in N_i \rightarrow n \notin N_j$ applies.

2.3 Target Platform Complexity

This section discusses problems arising from the complexity of the target platform which consists of a hardware component, the target hard- and firmware, and a software component, the operating system that runs on the target hardware and performs operations like input/output, interprocess communication, task management,

memory management and more. It is outside the scope of this thesis to fully examine the influence of the target environment on the execution-time but it it is necessary to know the effects caused by the target environment and how to prevent them from interfering with the execution-time analysis.

2.3.1 Influences of the Operating System

In general operating systems (*OS*) provide an interface between hardware resources, the external environment, which is often the user, and the application software running on the computer system [Sta04, Tan01]. Real-time embedded systems are different since the range of possible input values is known a priori in most cases and the system resources tend to be very limited for maximum cost-effectiveness. To maximize the utilization of the embedded system the application runs without an operating system in some cases, giving it full control over all resources. This approach has been followed for a long time in small isolated Electronic Control Units (*ECU*s) like engine control or similar tasks. As applications tend to become networked, like for instance x-by-wire applications in the automotive domain, a single networked control device is no longer self-reliant but needs to transfer information from and to other computing nodes in the network. It can also be noted, that it is desired by the manufacturers to integrate an increasing number of functions within a single device. A convenient way of adding functionality is to add independent tasks which provide this functionality. Real-time operating systems like RTAI, QNX®, TTP-OS® provide the required functionality, and can even be certified for safety-critical applications like LynxOS® or TTP-OS®. The following subsections examine individual services provided by common operating systems.

Blocking System Calls

Blocking system calls suspend the execution of a task until a specific event occurs or an IO operation is finished. Due to their unpredictability they should not be used in real-time application programming and are not supported by the WCET analysis tool prototype presented in this work.

Concurrency and Interprocess Communication

The operating system is used to ensure isolation of concurrent tasks performing individual functions on an ECU, to allow synchronization and communication between these tasks, and to perform scheduling, deciding when and how long the controlled tasks run. On some systems the OS provides also memory protection, preventing a task to access the memory assigned to another task.

Several problems arise from the concurrent execution of tasks: First, switching between tasks is a very expensive operation, requiring a considerable overhead in form of calculation time [LDS07, DCC07]. Additionally, schedulers for RTS have

to consider not only the performance of the tasks but also the timeliness of their completion. A basic prerequisite for real-time scheduling is therefore a correct WCET bound of the individual tasks running on the ECU. This is a requirement for both static and dynamic scheduling.

Second, the OS has to provide means for the application to allow synchronization and communication between individual tasks. In conventional OSs these functions are realized as blocking system calls which is bad practice for RTS [LLS+07, chapter 2.4], especially if the maximum blocking duration cannot be exactly predicted. Therefore some real-time systems use the concept of non-blocking tasks, in which communication channels are defined at the design time of the system. The concept of time-triggered communication [Kop98, KB03] is used widely in the automotive and avionic domain, for instance in FlexRay™ [Fle], TTP™ [TTP], TTCAN™ and LIN™ [Lin]. The basic idea of time-triggered communication is that a communication schedule defines exactly, when a message is to be sent or received. In a typical distributed real-time system all input data is stored within a portion of local memory before a task is activated. The task writes its output to a defined memory location and after the task terminates, the operating system transmits the data over the communication network. In many cases the communication within a single ECU uses the same mechanism, thus eliminating the need for other communication channels between tasks.

Communication between Network Nodes

As described in the previous section, time-triggered protocols are the means for providing communication between different nodes on a network for real-time applications. Communication protocols like the upcoming TT-Ethernet™, which adds a time-triggered communication layer on top of a standard ethernet, provide the communication functions found in a typical standard OS. However, these functions should not be used in real-time tasks since their timing behavior is unpredictible.

Interrupts and Signals

Interrupts may be caused by the hardware as well as the software partition of the target platform. The operating system is responsible for the correct handling of interrupts by providing *Interrupt Service Routines (ISR)* which suspend the current task and are executed when an interrupt request arrives. When a running application is suspended by an ISR additional control-flow is generated. The running task is not only delayed by the execution of the ISR, which can take relatively long when a context switch is performed, but additionally the internal hardware state, especially the cache and the pipeline of the CPU, is altered. Allowing interrupts anywhere makes application impossible to predict, therefore interrupts are forbidden by the task model used in this work (see below). An approach where interrupt handling is delayed until the execution of a real-time task has been finished is presented in [JSK+07].

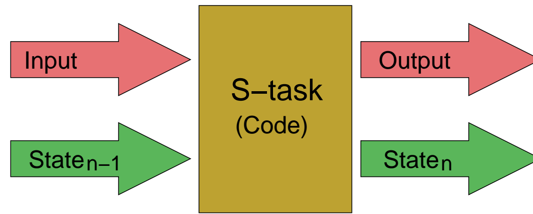


Figure 2.4: The S-task model

The Simple-Task (S-task) Model

The work in this thesis assumes, that a running task is not interrupted by the operating system nor uses blocking system calls, this means, a task runs uninterrupted from the time it is executed by the OS until it terminates. This model of a task is called a *simple task* (S-task) [Kop97]. Figure 2.4 shows a graphical representation of the *simple task* model. The S-task depends on the state from the previous execution $\text{state}_{\text{n-1}}$ and input data. The S-task generates the state for the next execution state_{n} and output data. Therefore the execution-time of the S-task is fully determined by the code of the S-task, the input and the state from the prior execution.

The concept of the simple task might appear as a crude simplification. However, this model applies for the majority of real-time systems, since the communication is performed by the operating system at the beginning and at the end of the task and input/output operations like reading or writing a register or hardware port are generally non-blocking.

As counterpart to the simple task Kopetz [Kop97] also introduces the concept of a *complex task* (C-task) which may contain (blocking) synchronization statements like semaphore wait operations or a blocking read or write operation. It is important to note that "simple" or "complex" does not state anything about the complexity of the performed computations.

2.3.2 Influences of the Compiler

The compiler that is used to transform the source into an executable for the target platform has also an influence on the execution-time of the compiled application. The compiler can perform optimizations which alter the control-flow of a program and thereby remove or introduce control-flow paths. For example, the shift operators « and » are likely realized as loops on simple architectures where there is no shift opcode with variable shift operands. It is difficult to consider all compiler specific CFG transformations. Therefore we introduce the prerequisite, that the CFG may not be altered by the compiler (see section 3.4.1).

The following sections give a short overview of the most common compiler (hardware independent) and code generator (hardware dependent) optimization techniques.

Loop Optimizations

Loop optimizations are optimizations on the loop entry/exit, the condition check or the loop body. As loops are typically often iterated the performace boost of these optimizations can be high, rising with the number of iterations. Loop optimizations are performed in the middleend of the compiler.

Loop Unrolling decreases the number of required loop condition tests and jumps by inserting multiple duplicates of the loop body. When the number of iterations is known at compile time the loop may be completely unrolled. Care must be taken to avoid negative effects caused by the increased code size, especially when an instruction cache or shared cache is used. A popular example for explicit software-based loop unrolling which shows also some oddities of C is the infamous Duff's device[Duf83]. This optimization has heavy influence on the control-flow and should therefore be turned of for measurement-based WCET analysis.

Loop Inversion transforms a *while* loop into a *do/while* loop embedded into an *if* statement reducing the number of required jumps by two assumed that the loop body is at least executed once. Like loop unrolling this technique alters the control-flow and should be turned off.

Induction Variable Analysis is used to identify variables that depend on the number of loop iterations and perform strength reduction (see below) on them.

Data Flow Optimizations

Data flow optimizations change the way how expressions are evaluated. For these optimization techniques a data flow analysis is required. A simple subset of data flow analysis can be implemented to perform a subset of these optimizations. Data flow optimizations are performed in the middleend of the compiler.

Constant Folding and Propagation replaces constants like $4*4$ or $arcsin(1)$ with their values (16, $\pi/2$) at compile time. If the compiler is known to perform this optimization it does not interfer with WCET analysis.

Common Subexpression Elimination (CSE) identifies duplicate subexpressions that can be substituted by a temporary variable. An expressinn like "`q=(a+b+1)/(a+b-1);`" can be transformed to "`t1=a+b; q=(t1+1)/(t1-1);`" eliminating one unnecessary addition. This optimization should be safe for WCET analysis unless it is applied to logical AND or OR expressions. This optimization can change the control-flow of the application, especially when logical AND or OR, or `?:`-operators are involved.

Code Generator Optimizations

Code generator Optimizations require knowledge of the target hardware. They try to optimize the generated code for this architecture. Code generators are able to use special features on the target hardware, like the different SSE_x unit implementations on individual Intel® x86 series processors. Since these optimizations depend strongly on the target architecture, they are performed in the compiler backend.

Register Allocation is used to keep the most frequently used variables in CPU registers for fast access.

Instruction Selection and Scheduling is used to select the most effective instructions for the operations, which shall be performed, to optimize the utilization of the functional units of the CPU and keep the pipeline filled. With the increasing use of transparent hardware optimization techniques these optimization become less effective because the hardware interferes with the optimizations performed by the compiler. Since the compiler has a better global view of the application the utilization of functional units may be better than with hardware optimization.

Other optimizations

This list includes optimization techniques that cannot be assigned to the categories above and which may be either platform independent or not.

Inline Expansion inserts the function bodies instead of function calls. As function calls usually include stack operations and two jump operations, possibly causing pipeline stalls, they are rather expensive operations in terms of execution-time. The control-flow of the object code is changed by this optimization. If the function is expanded before compilation in the source code, like it is done in this thesis, the control-flow in source and object code should not differ.

Dead Code Elimination removes code that is never executed. This is done by reachability analysis. Under normal conditions this should not change the execution-time but when cache is used there might be effects on the execution-time.

Reduction of Cache Collisions 2.3.4 tries to place data structures of functions in a way to avoid cache conflicts. To perform this optimization detailed knowledge about the cache organization on the target hardware as well and about code relocation techniques on the target operating system is required.

Strength Reduction is a compiler optimization where a costly operation is replaced with an equivalent, but less expensive operation. This is often used on binary operators where one of the operands is a constant. As an example `y = x / 2` can be replaced by `y = x » 1`. The control-flow is not affected by this optimization as long as no logical AND or OR, or the `?:`-operator is changed.

The list of optimizations is far from complete. Only the most important optimization techniques have been shown. A complete list of optimizations and how they may change the control-flow can be found in [Kir03].

2.3.3 Influence of the Target Hardware

The hardware portion of presents target systems is even more difficult to handle, than the software portion. This is mainly because of the fact that it cannot be observed what happens in the hardware. An object code file or an OS can be analyzed directly but to examine the processes taking place in the hardware an exact model of the hardware or a custom implementation with a debugging interface is required.

Hardware models are difficult to create, especially for modern processors which are superscalar and have caches, pipelines, branch prediction, out of order execution and speculative execution amongst other features. Some processor families use microcode internally and even allow this microcode to be patched[MP02, SNC^{+}07]. And last the user manual for the target hardware is, due to intellectual property concerns, not always complete or even erroneous. Therefore the effort to create exact hardware models is high and sometimes requires reverse engineering, which is forbidden under the legislation of some countries.

Debugging interfaces allow amongst other features to set breakpoints, to read registers and to trace the execution of a program in real-time. Hardware debugging is usefull for simple architectures but it does not allow to examine the contents of caches or the processor pipeline and can therefore not be a replacement for proper hardware models.

Runtime measurements are also a way to get information about the target. In this process the target hardware is used instead of using a model of it [KWRP05]. To increase the reliability of the measurements a pre-defined state has to be established which can be done by flushing the cache and stalling the pipeline, which causes the performance to drop. Measurements performed without this precautions are less reliable.

The following sections focus on individual hardware optimization techniques. It will be discussed why these techniques make the analysis more complex and how they effect the execution-time.

2.3.4 Hardware Optimization Techniques

The Goal of all Hardware optimization techniques is to reduce the average execution-time, which is accomplished in the majority of the cases. This means that the execution is optimized for the frequent executed paths and that the less frequent executed paths are likely to suffer from increased execution-time. As the WCET occurs generally during the execution of less frequent paths the WCET is likely to be increased by some of the discussed hardware optimization techniques.

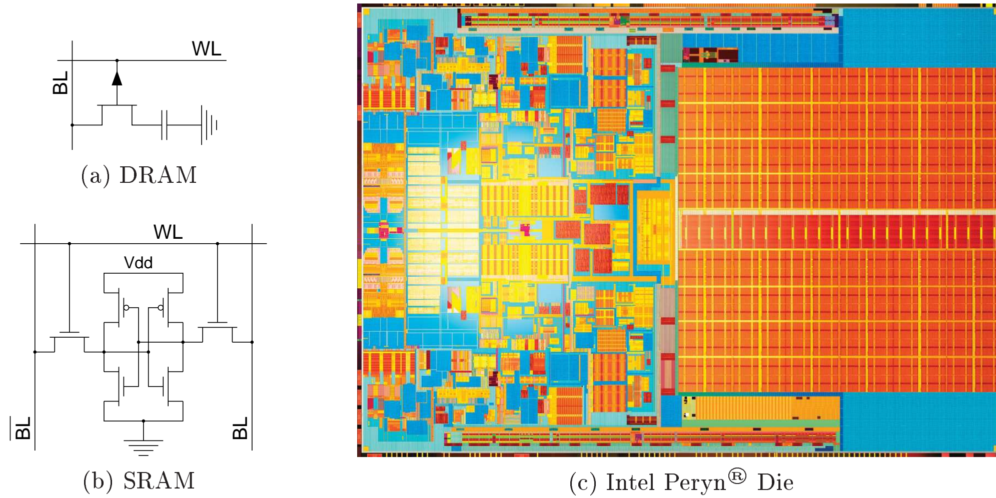
Historically the emergence of hardware optimization techniques was founded of the lack of optimizing compilers. Since compilers did not support the features of new processors the hardware developer tried to make optimizations independent from compiler support. According to [Fis88] this brings a lot of disadvantages: First things done at compile time are only done once. Even if a hardware optimization does not increase computation time because it runs concurrently, it still needs die space and power. Second, the analysis performed during execution cannot be as thourough as when performed during compilation. It is also not possible to adjust optimization settings for different kinds of problems. And third, little hardware is required to support optimizations that are performed during compile time. However, [Fis88] points out the single advantage of hardware optimizations: Values which cannot be determined or are very hard to determine at compile time are available in the hardware during execution. This includes values of variables, pointers and branch targets.

The next sections discuss common hardware optimizations and how they affect execution-time analysis. The common characteristics of hardware optimizations is that they all, with few exceptions, introduce additional complexity making execution-time analysis harder.

Caches

Caches are one of the oldest performance enhancements dating back to the 1960s where they were used in mainframes. Today caches are used to build up memory hierarchies where small but fast areas of memory are located near or in the CPU and the large but slow main memory is accessed through them. This design usually decreases the average access time and reduces the wait-states of the CPU. The first level of cache, the L1 cache, a modern computer system typically provides, is a small on-chip instruction- and data-cache, which may be unified or separated, and has a size of $2\,KiB$ to $64\,KiB$. The L2 cache, which is also on-chip in most cases ranges from $128\,Kib$ to $1\,MiB$. The L3 cache, if it exists, ranges from about $2\,MiB$ to $256\,MiB$.

The reason why the caches are kept relatively small can be seen in figure 2.5. Dynamic Random Access Memory (*DRAM*), which is used for the main memory of modern systems, requires only a single transistor which requires a size of 6-10 $F62$, where F is the Featuresize and typically 45-120 nm. Reading is done by applying a voltage to the word line (WL). Depending on the charge state of the capacitor a logic “0” or a logic “1” can be observed at the bit line (BL). The disadvantage is, that the capacitor suffers from self-discharge and has to be charged at least each $64\,ms$ according to the Joint Electron Device Engineering Councils (*JEDEC*), a process during which the memory cannot be accessed. When writing the capacitor has to be charged or discharged, a process that requires a few ns. Static Random Access Memory (*SRAM*) does not have any capacities, therefore it does not need refresh cycles and the read/ write operations are considerably quicker. The disadvantage of SRAM is that it requires six transistors and it requires an area of $140\,F^2$ for a



(a) DRAM

(b) SRAM

(c) Intel Peryn® Die

Figure 2.5: DRAM vs. SRAM Cell and Processor Die with SRAM Cache

single bit, which is 23 times more than for DRAM. In figure 2.5(c) a die of a modern processor with 2 MiB of L2 cache, the Intel Peryn®, is shown. It can be seen that the cache, which is the big area with the regular pattern at the right side, uses almost half the die area. Therefore it is obvious that the consumed die area is the main size limitation for the cache.

The logical architecture of a single cache level can be *direct mapped*, where a single memory location can only be mapped on a single cache line,*n-way set associative*, where a memory location can be mapped on one out of n possible cache lines, or *direct associative*, where a memory location can be mapped to any cache line [HP06, PH98]. In combination with caches Transaction Lookaside Buffers (TLB) can also be used. TLBs are used as a cache for the Memory Management Unit (MMU) which translates the virtual addresses used in programs to physical addresses located in the RAM [HP06, PH98], introducing yet another level of redirection.

Figure 2.6 shows the memory hierarchies in the AMD Opteron® processor family. Instruction fetch involves the L1 cache and TLB, the L2 cache and TLB, the system chip memory crossbar and the system DRAM, depending if the instruction can be found in the L1 or L2 cache. A similar data flow occurs when reading or writing memory. For cache analysis, which is a subtask of static time analysis, this introduces a vast complexity. As an result there are numerous works that focus on this area. An overview of these works is given in section 6.5.

The work presented is this thesis uses the ARM® architecture for the performed execution-time measurements. When running from the RAM there is no cache present since the RAM is fast enough to allow single-cycle memory operations. Therefore no cache analysis is performed in the presented work.

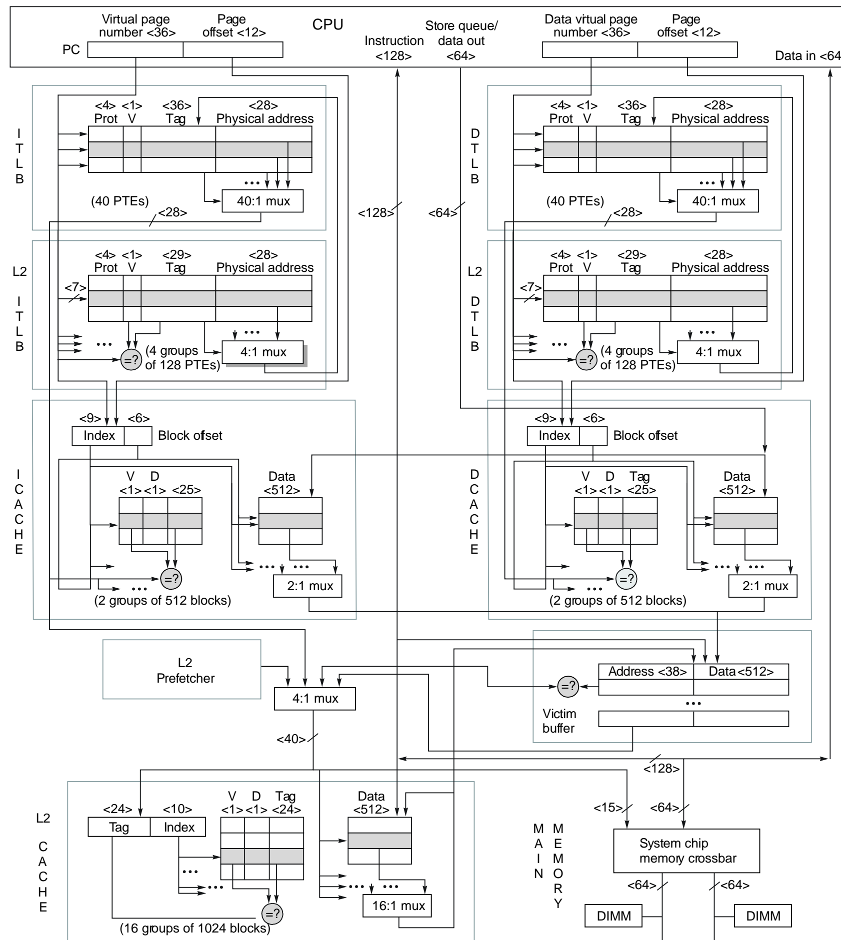


Figure 2.6: AMD Opteron® Memory Hierarchy [HP06]

Pipelines

Pipelines are used to divide complex operations into a series of simple operations which can be completed within a shorter clock cycle, thus allowing higher clock frequencies. The individual pipeline stages are separated by registers which are synchronously updated on each clock cycle and feed the results of a pipeline stage to the next stage, a common concept known as synchronous logic. Modern processors can have 10-20 pipeline stages, the Intel Pentium® Prescott and Cedar Mill has a pipeline with 31 stages.

Figure 2.7 shows a simplified MIPS pipeline consisting of five states. The real pipeline is more complex since it adds datapaths as shortcuts between individual stages and has to deal with numerous exceptions, i.e. when a command uses a result of a previous command it can be taken directly from the Arithmetic Logic Unit (ALU) output instead of waiting for it for three additional pipeline stages.

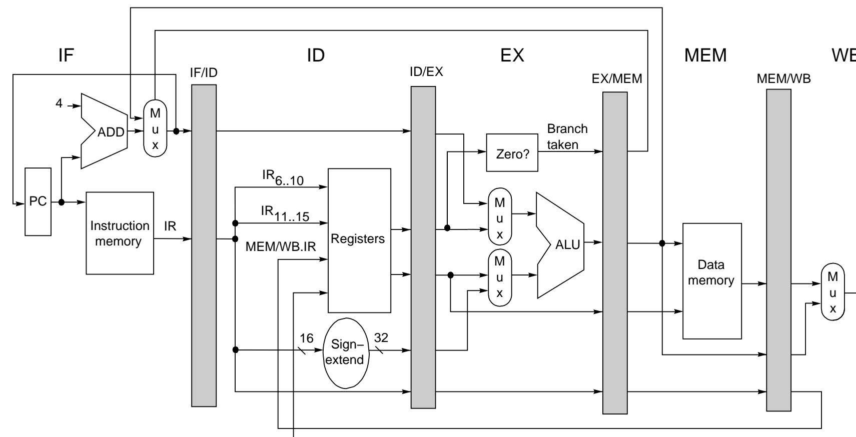


Figure 2.7: Simple MIPS Pipeline [HP06]

1. *Instruction Fetch* (IF)
   This stage fetches the next instruction from the memory and stores it in a register. The program counter is also increased in this stage.

2. *Instruction Decode*(ID)
   This stage decodes the instruction and prepares the registers as well as the immediate value that may be contained in the lower 16 bits of the instruction. This process is also known as dispatching, especially is the architecture has multiple parallel execution units.

3. *Execute* (EX)
   Executes the specified function which may be
   adding an immediate value to a register
   $\text{ALU} \leftarrow \text{A} + \text{Imm};$
   performing a function with the two registers
   $\text{ALU} \leftarrow \text{A}\, \textit{func}\, \text{B};$
   performing an operation combining a register and an immediate value
   $\text{ALU} \leftarrow \text{A}\, \textit{op}\, \text{B};$
   adding an immediate value to the new program counter
   $\text{ALU} \leftarrow \text{NPC} + (\text{Imm} << 2);$

4. *Memory Access* and *Branch Completion* (MEM)
   Performs load or store operations, reading register contents from the memory or writing registers to the memory, or performs a branch, replacing the program counter PC with the output of the Arithmetic Logic Unit (ALU).

5. *Register Write Back* (WB)
   Writes the result to the register file. The written value can be the ALU output or the result of a memory read operation.

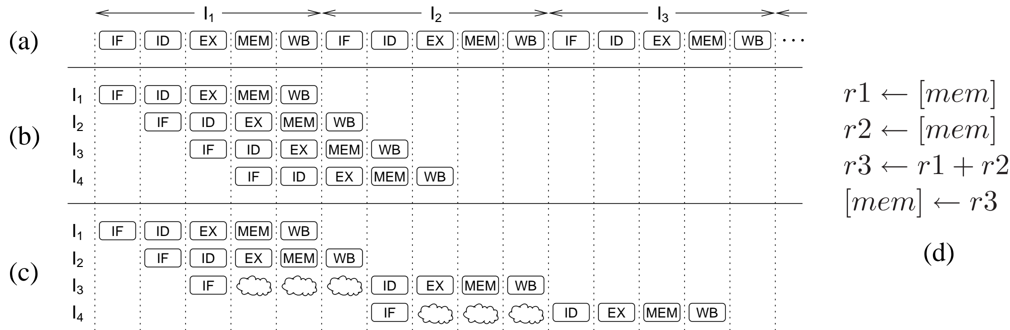


Figure 2.8: Pipelined Execution of Instructions [AEBER04]: (a) Non-pipelined, (b) Idealized Pipeline, (c) Simple MIPS Pipeline, (d) Executed Commands

Figure 2.8 shows the execution of a short command sequence shown in subfigure d on three different architectures. Subfigure a shows the execution in a simple non-pipelined architecture where all processing steps have to be executed in series, requiring a total 20 clock cycles. Subfigure b shows the execution of the same commands on a pipelined architecture with an idealized pipeline using eight clock cycles. Subfigure c depicts the execution of (d) on the MIPS pipeline as seen in figure 2.7. The decoding of I_3 has to be delayed until the Register Write Back stage of I_2 is finished since I_3 depends on results written to the register file from I_2. For the same reason I_4 has to be delayed until I_3 is finished. During these delays, the some pipeline stages are empty and "bubbles" are inserted. The total execution-time is 14 clock cycles. More advanced pipelines provide mechanisms like forwarding to reduce the described effect to a minimum[HP06, PH98].

Some situations, called pipeline hazards, enforce the delay of individual pipeline stages of instructions, thus reducing the performance from the ideal performance as shown above, a situation denoted as pipeline stall. Pipeline Hazards are categorized in three different categories.

1. Structural hazards
   This kind of hazard arises from resource conflicts when different stages of the pipeline require the same resource.

2. Data hazards
   Data Hazards arise when an instruction depends on results of a previous execution and needs to wait for these results to become available.

3. Control hazards
   Control hazards arise from branch or jump instructions when the program counter PC is changed by the ALU.

Pipelining allows instruction level parallelism and superscalar execution. Early processors were scalar, which means they allowed only the dispatching of a single instruction per clock cycle. Todays processors are superscalar and allow the dispatching

and completion of multiple instructions at the same time. This can be achieved by having multiple parallel execution units, i.e. two integer units and a floating point unit, which can operate in parallel. Hyperthreading, which is an Intel™ proprietary implementation of simultaneous multithreading, allows the simultaneous execution of different tasks on individual functional units of the processor. A floating-point intensive application could run simultaneously with a memory intensive application without interference at the same processor.

Based on the degree of instruction level parallelism we can classify pipelined architectures in different subtypes [Wen06]:

1. *Simple scalar pipelines* consist of a single pipeline, e.g., as depicted in figure 2.8(a). The functionality of each stage may vary. The key characteristics of simple scalar pipelines are that there is only one way through the pipeline and thus a maximum of one instruction per cycle (IPC) can be achieved.

2. *Scalar pipelines* extend simple scalar pipelines by adding additional execution units at some stages, e.g., there are two integer units on the execute stage. However, at most one instruction can be issued per cycle. Multiple execution units make sense, especially when the latencies of instructions can be multiple cycles. For this pipeline type the condition IPC ≤ 1 holds.

3. *Superscalar pipelines* allow issuing more than one instruction at a time, i.e., IPC can be greater than 1. These machines are called multiple issue machines. Which instructions are issued at the same time is determined statically (statically scheduled at compile time).

4. *Dynamically scheduled pipelines* allow the online determination of the instruction scheduling. This is the most complex type of pipeline because it allows out-of-order execution of instructions.

As pointed out in [Wen06] this list neglects special purpose hardware designs like digital signal processors (DSPs) and vector machines because these types are not further relevant for the systems addressed by this thesis.

Branch Prediction

As pointed out above, conditional branches cause pipeline stalls. If the branching condition is already available, the correct decision wether the branch is taken or not can be forwarded to the instruction fetch unit. Otherwise a control hazard will occur. Speculative execution is a simple way to reduce pipeline stalls. The processor predicts if branches are taken or not and continues the execution directly at the correct point. If the prediction proves to be incorrect, all computation past the branch point is discarded, if it is right execution continues without any pipeline penalties.

The simplest form of branch prediction is to assume that the branch is never taken, which is a form of static branch prediction. The compiler optimizes the code

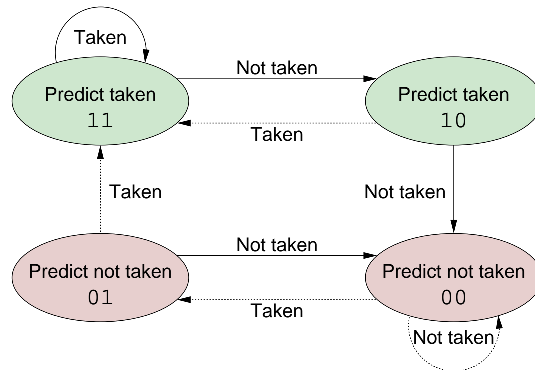


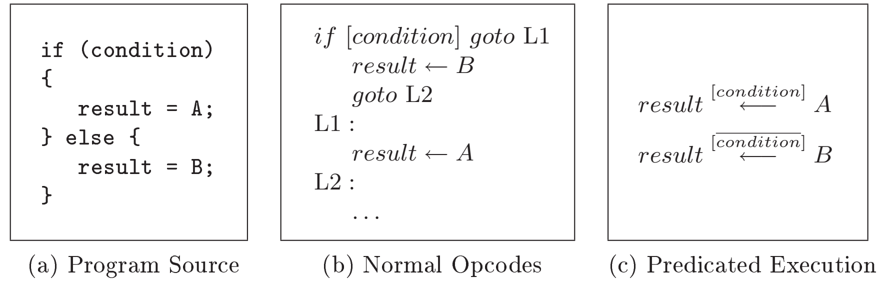
Figure 2.9: 2-bit Branch Prediction [HP06]

generation so that the branch will not be taken in most of the cases thus increasing the number of correctly guessed branches. An other form of static branch prediction is to assume that backward branches are always taken because they are likely to be loop condition checks and forward branches are never taken. The later approach works better if the compiler is not aware of the branch prediction.

Modern processors use dynamic branch prediction. An example for a simple dynamic branch prediction logic is a 2-bit branch prediction. The branch prediction uses a *branch-prediction buffer* or *branch history table*, which is an area of memory that holds the 2-bit state of the branch prediction state machine shown in figure 2.9 and is indexed through the lower bits of the branch instructions address. The prediction is taken from the observed state of the branch prediction state machine and after the branch condition is evaluated, the result is executed feed back into the state machine in form of a state change. The prediction rates are well above 90% for five out of seven of the SPECint benchmarks[PSR92].

Predicated Execution

Predicated execution is another way of reducing control hazards. Observing a simple piece of source code like in figure 2.10(a) we assume that it is translated to something like (b). However, this few lines of code generate two jumps, from which one of them is always executed. Predicated execution provides a very elegant solution for this problem by executing short sections of code conditionally instead of jumping over them. The code shown in (a) is translated into two subsequent assignments as shown in figure 2.10(c). Both assignments are executed within the CPU but depending on the value of the predicate, which is denoted as condition, the assignments are either committed or discarded during the register write back stage of the pipeline. Thus the behavior and the internal state of the CPU are completely independent of the predicate, unless of course a operation which modifies the CPU flag register is predicated. Predicated execution is a important foundation for single path conversion which will be presented in section 6.2.2. An important disadvantage of predicated execution is that each instruction requires an additional bit field where



(a) Program Source (b) Normal Opcodes (c) Predicated Execution

Figure 2.10: Normal Opcodes and Predicated Execution Opcodes

the predicates are stored, which is especially a problem in small embedded systems. In contrast to most of the other optimizations techniques which introduce additional complexity, this technique reduces the number of execution paths by eliminating conditional jumps. Processors which provide predicated execution include the Intel® IA-64™ Architecture and the ARM™ Architecture. The ARM™ Thumb-2® extensions provides even a special instruction which does nothing but to provide the predicates for the following four instructions [Phe03].

Instruction Latency Jitter

Some instructions, especially mathematical operations like multiplication, division and most of the floating point operations have inconstant execution-time, depending on the value of the operands. Figure 2.11 shows an overview of selected instructions and their execution-time in clock cycles on differnet Processors of the Intel® x86 Processor Family. The operands for all operations are registers, except for the `CMPS` opcode where this is not possible. It can be seen that some operations, like most of the floating point operations, have a highly variable execution latency while other operations like the string search are even more complicated since they are basically implemented as loops.

It has already been mentioned in section 2.3.2 that C shift operators may be implemented as loops on simple architectures, but even on more sophisticated CPUs the execution-time may vary within a wide range as long as they do not have a barrel shifter unit. Even if they have one, rotate commands that include the carry bit are often unable to take advantage of the barrel shifter unit.

The current execution-time analysis prototype does not take different execution-times into account. Possible workarounds are to restrict the use of functions with variable jitter to constant operands or to add an additional time budget for these instructions. The experiments conducted in this work avoid variable latency executions.

If variable latency instructions are used, the required margin in processor cycles can be calculated as $T_{jitter} = T_{max} - T_{min}$, where T_{max} and T_{min} are the maximum respectively minimum instruction latency in CPU cycles. When using the assignment

Opcode	Interpretation	80386/7	80486	Pentium
`ADD/SUB`	Integer Addition/Subtraction	2	1	1
`MUL`	Unsigned Multiply	9-38	13-42	10
`DIV`	Unsigned Divide	38	40	41
`RCL`	Rotate Bits Left with CF	9	8-30	7-24
`ROL`	Rotate Bits Left	3	3	4
`BSF`	Bit Scan Forward	10+3n	6-42	6-42
`BSR`	Bit Scan Reverse	10+3n	7-104	7-71
`CMPS`	Compare Strings	5+9n	7+7n	9+4n
`FADD/FSUB`	Floating Point Add/Sub	23-34	8-20	1
`FMUL/FDIV`	FP Multiply/Divide	88-91	73	39
`FYL2X`	Compute $Y * \log 2(x)$	120-538	196-329	22-111
`F2XM1`	Compute $2^x - 1$	211-476	140-279	13-57
`FSIN`	Sine	122-771	257-354	16-126

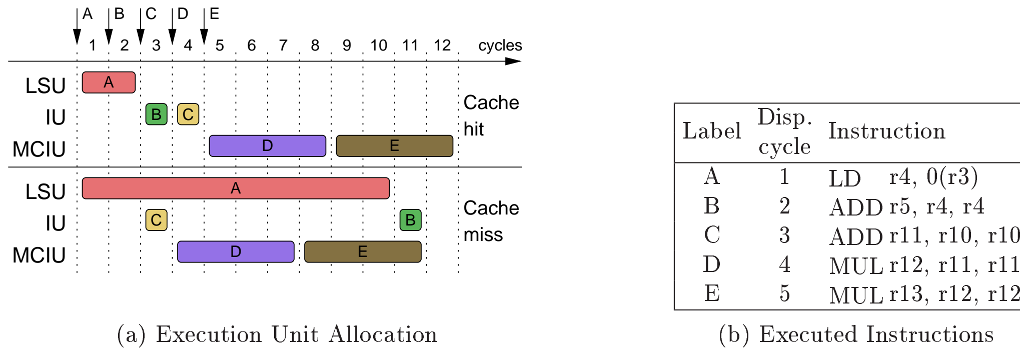
Figure 2.11: Variable Execution Times on the Intel® x86 Processor Family

`y = sin(x)` on an Intel® i80486 we would have to add a safety margin of $354 - 257 = 97$ cycles.

Timing Anomalies

A negative impact on the global execution-time caused by a local optimization is called a timing anomaly. Timing anomalies are in most cases the result of the dynamic allocation of resources during execution. Like noted in section 2.3.4 the hardware has only a local view and can therefore only perform local optimization. In some cases this lack of global information can lead to an overall performance loss. Timing anomalies are a relatively young area of research. The term “Timing Anomalies” was introduced in context of task scheduling in [GG69] and first used for instruction scheduling and timing analysis of single tasks in [LS99b]. According to Lundqvist a timing anomaly occurs when a locally decreased latency results in an globally increased latency or a locally increased latency results in a globally decreased latency [LS99b]:

> Consider the execution of a sequence of instructions. Let us study two different cases where the latency of the first instruction is modified. In the first case, the latency is increased by i clock cycles. In the second case, the latency is decreased by d cycles. Let C be the future change in execution-time resulting from the increase or decrease of the latency. Then:
>
> **Definition** *Timing Anomaly* [LS99b]
> A timing anomaly is a situation where, in the first case, $C > i$ or $C < 0$, or in the second case, $C < -d$ or $C > 0$.
>
> That is, if C is guaranteed to be in the interval: $0 \leq C \leq i$ in the first case or, $d \leq C \leq 0$ in the second case, we have no timing anomalies.



(a) Execution Unit Allocation

Label	Disp. cycle	Instruction
A	1	LD r4, 0(r3)
B	2	ADD r5, r4, r4
C	3	ADD r11, r10, r10
D	4	MUL r12, r11, r11
E	5	MUL r13, r12, r12

(b) Executed Instructions

Figure 2.12: A Timing Anomalie Example from [LS99b]

Figure 2.12 shows an example of a timing anomaly taken from [LS99b]. Opcode A depends on data from B and E depends on D which depends on data from C. In the first case the read operation takes advantage of the cache. After A ic completed B is issued to the integer unit (IU). This delays the execution of C and as a result D and E. In the second case, when a cache miss is encountered, C is issued immediately to the IU while B waits still in the reservation station of the integer unit. When A finishes B is issued to the IU. This example shows how an increased execution-time for a single operation (8 cycles) can result in a reduced execution-time for a series of operations (1 cycle).

Timing anomalies are a complex topic and currently the effort in the scientific community is to find a suitable definition for timing anomalies [Lun02, RWT$^+$06, KKP09]. The question whether timing anomalies can occur on a specific hardware architecture is even more difficult to answer generally. Wenzel defines the Resource Allocation Criterion (RAC) in [Wen03] where he shows that a resource allocation conflict is a necessary precondition for timing anomalies, but the underlying hardware model is very restricted since it defines only functional processor units as resources and neglects timing anomalies caused by different cache replacement strategies [The04] or specular execution [RWT$^+$06]. Section 6.6 gives an overview of recent publications on timing anomalies.

2.4 Software Complexity

Naturally, the complexity of the analysis depends not only on compiler-, operating system- or target specific elements but also on the complexity of the application which depends on the functionality of the underlying application model and requirements. This section gives a short overview how complexity can be measured using different software metrics and how coding guidelines can be applied to increase the reliability, readability and maintainability of applications.

2.4.1 Software Metrics

Software Metrics are used to enumerate the size, complexity, cost, reusability, quality and other aspects of software components. In the context of this work software metrics are used to determine if the timing behavior of a given application can be analyzed with reasonable effort using the measurement-based approach. The metrics used in this work, lines of code (LOC), Number of basic blocks and path coverage (number of end-to-end paths) are described below. In addition the size of the source file is sometimes used as a measurement of complexity. However, the validity of this number is very low.

Lines Of Code (LOC)

The source code lines are a software metric which is widely used and in this work it is used to give a quick estimate about the size of an application. There are two ways to measure the LOC: The first is to count the physical lines of code (which can be done using a text editor). The second method is to count the number of logical and whitespace lines, where each logical line is a single statement. Additionally the comment lines can optionally be included in the number of LOC. The term LOC refers to physical lines of code including comments and empty lines when used in this thesis. The problem using LOC, especially for giving an estimate of application complexity, is that the code lines do not take the expressivity of the source language as well as the overhead generated by code generators or the application languae syntax.

Number of Basic Blocks (BB)

The number of basic blocks denotes the numner of conjoined sequences of expression statements without any control-flow decision or control-flow merges between them. The number of BBs can provide a better estimation of application complexity than the number of source code lines (LOC).

Cyclomatic Complexity

The cyclomatic complexity measures the linear independent paths through an application and gives a good estimate about the overall complexity of an application. Only linear independent paths are considered. The cyclomatic complexity can easily be evaluated when the number of basic blocks, connection between basic blocks and exit points is known. However, a drawback is that the cyclomatic complexity is between the number of decision branches and the total number of end-to-end paths Π. Therefore the cyclomatic complexity can lead to an underestimation of the timing analysis effort for an application since this effort depends on Π.

Path Coverage

Full path coverage is primarily a criteria for software testing but it can also be used for describing code complexity. Path coverage gives the total number of distinct end-to-end paths through an application. Since the proposed WCET analysis technique relies heavily on full path coverage this number is used to give an estimation of the complexity and the analysis effort for a given application. Special care has to be taken with loops. When encountering a loop, the WCET analysis tool counts the paths in the loop body, and adds one to it if the loop check is at the top of the loop. The actual number of paths would be the iteration maximum loop times the paths in the loop body. However, since this number is unknown at the beginning of the analysis, and remains unknown for certain types of loops, the proposed solution gives a quick and accurate estimate of the application complexity.

2.4.2 Coding Guidelines

Coding guidelines are used in many companies to ensure the readability and reusability of source code. The coding guidelines essential for safety-critical applications are the Motor Industry Software Reliability Association (MISRA) guidelines [MIS96, MIS98, MIS04] in the automotive and DO-178B [RB92] and Avionics Application Software Standard Interface (ARINC 653) in the avionic domain. The following paragraphs briefly illustrate the impact of those guidelines on timing analysis.

The MISRA guidelines define a subset of C that should be used for industrial applications and consists of a set of rules to which an application has to comply. These rules are checked statically by means of a MISRA rule checker or a MISRA-aware compiler. The MISRA guidelines relate to WCET analysis as they disallow the use of dynamic memory, variable number of function arguments, unreachable code and more C language features that make WCET analysis difficult, however they do neither mention the concept of time nor how timing analysis shall be performed.

The DO-178B standard is used for safety-critical airborne applications. However, the DO-178B guidelines focus on the environment under which software is developed, which includes the software life cycle as well as process activities and design considerations and their monitoring. The application itself is not focused in the DO-178B specification.

The Avionics Application Software Standard Interface (ARINC 653) describes how components of an integrated modular system should be separated and individually tested. Execution time analysis is mentioned briefly but not described in detail [APE03].

A detailed description, how the above mentioned guidelines relate to timing analysis has been published in [WKS+05].

2.5 Summary

In this chapter, the different C dialects as well as unique and problematic features of C have been pointed out. Further, the basic concepts of compiler construction have been introduced and it has been shown how the compiler, the target OS and the target hardware can influence and increase the complexity of timing analysis. Last it has been shown, how code metrics can be used to give a coarse estimation of the complexity of an application and how industrial coding guidelines interfere with the timing analysis process.

Chapter 3

Execution Time Analysis

This chapter explains the importance of (worst-case) execution-time analysis and introduces different approaches to perform timing analysis. The main focus is on the hybrid timing analysis for which an analysis framework prototype is implemented as a part of this thesis.

3.1 Why Timing Analysis

A real-time system (*RTS*) is a system in which the correctness of a calculation not only depends on the value of the result but also on the time when it is finished. RTS are subclassified in *soft real-time* systems, which are primarily used for non safety-critical systems such as mobile phones or the update management of flight planes for commercial air lines, and *hard real-time* systems, which are used for safety-critical applications, where a failure would result in massive damage of property or the loss of human life. Typical applications of hard real-time systems include automotive applications such as X-by-wire or medical systems. In hard real-time systems, computations are useless if the result is available too late. Soft real-time systems tolerate late results but respond with decreased quality of service. Figure 3.1 shows the application domain of real-time systems.

Another categorization of real-time systems is how the scheduling is performed. *Event triggered* RTS defines a set of tasks with given priorities. When an event arrives which has higher priority than the currently executed task, the task is interrupted

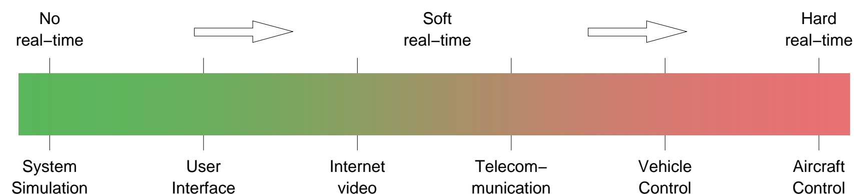


Figure 3.1: Timing Requirements [LLS$^+$07]

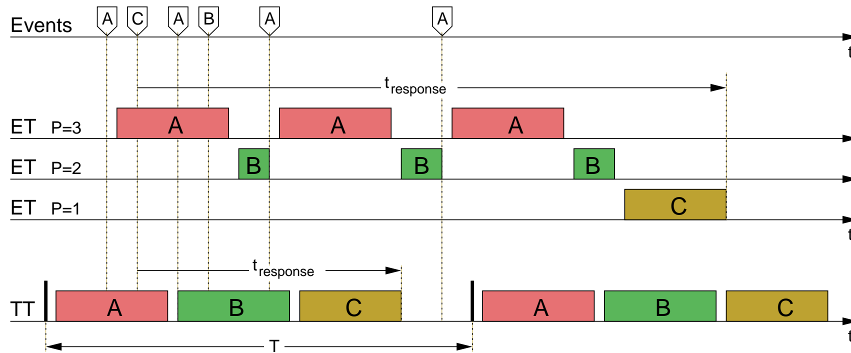


Figure 3.2: Response Time

and a task handling this particular event is executed. If the event has lower priority then it will be handled when the current task has finished and no higher priority task is scheduled. In some event-triggered RTS the priorities of the tasks can be dynamically changed to prevent the starvation of low priority tasks. *Time triggered* real-time systems define a fixed schedule at design time and the tasks running on the system are activated according to the pre-defined schedule. Some time-triggered RTS allow mode switches through which the system may change the predefined schedule to another predefined schedule. Distributed systems are characterized by a set of communication nodes which are interconnected using a communication network. The communication between nodes is either event or time-triggered according to the design of the nodes. Time triggered communication networks use cycle-based communication and send messages periodically resulting in a highly predictable system. Event triggered systems send messages only on demand and are considered more flexible. A disadvantage of event-triggered systems is that in the case of a fault there might be increased network traffic due to error handling which can lead to an overload and eventually a breakdown of the communication system. There are also communication systems that soften the borders between event and time-triggered communication. FlexRay [Fle] offers a time-triggered (static) and an event-triggered (dynamic) part in order to combine the advantages of both approaches, the predictability of the time-triggered systems and the flexibility of event-triggered systems.

Figure 3.2 shows the scheduling of three tasks on two different real-time systems. The first timeline shows the events as they are triggered from the outside world. Assuming event A is triggered by a switch which is bouncing. The next three lines show the execution of individual tasks with different priorities $P{=}x$ on the event-triggered system denoted with $ET\ P{=}x$ where higher numbers have higher priorities. Task A has the highest priority and gets executed on each event. Task B has lower priority than A and gets interrupted each time A is executed. Task C has the lowest priority and gets executed only when neither A nor B are scheduled. Thus the *response time* (RT) of task C, which is defined as $T_R = t_{reaction} - t_{event}$ gets very high and is eventually unbounded. The time-triggered RTS, shown in the bottom line of figure 3.2, uses fixed execution frames with a period of T, which is also called

cycle. Under the assumption that event C occurs on the least favorable moment after task C has been started and the event cannot be detected by the running instance of C the response time is bounded with $T_{R_{max}} = T + T_{Exec_{max}}(C)$. Some time-triggered RTS use the strategy to read inputs only at the beginning and write the outputs only at the end of a cycle. The values are stored in memory which the individual tasks access. This has the advantage that all task share the same view of the system state (increasing predictability [Kop97]) and that no input/output operations are necessary during a cycle. Additionally when using distributes real-time systems multiple tasks can share a communication frame. In this commonly used application the response time is bounded with $T_{R_{max}} \leq 2 * T$.

In the previous paragraph the expression $T_{Exec_{max}}(C)$ was used, which denotes the maximum possible execution-time of task C which is also called Worst-Case Execution Time (WCET) of C.

3.2 Worst-Case Execution Time (*WCET*)

When the term *WCET* is used in this thesis, it refers to the highest possible execution-time (ET) of a simple task as defined in section 2.3.1.

The *WCET* is one of the most important attribute of tasks when designing real-time systems because the scheduling depends on it. In fact the WCET is required to determine if a given task set can be scheduled on a system:

Definition 10 *Schedulability criterion [Wir01]*
A real-time system with a task set of n tasks having a worst-case execution-time of C_i and a period of T_i is schedulable on a system with p processors if

$$p \geq \sum_{i=1..n} \frac{C_i}{T_i}$$

The schedulability criterion is only a quick estimate since it neglects the time required for task switches and the core functions of the operating systems. For time-triggered RTS there is the additional condition that the WCET of each task must be lower than the duration of its execution slot.

Figure 3.3 shows the *execution-time profile* (ETP) of a simple task. The x-axis represents the execution-times of the task. As a unit for the execution-time *clock cycles*(CC) are often used since they are independent of clock frequency drifts and have a discrete value. Using absolute values like μs is not recommended because of the influence of the oscillator drift and because of the analog value which prevent an exact reproduction of the results. The value in the ordinate represents the frequency of the observed execution-time in relative or absolute values. The *average-case execution-time* (ACET) is the weighted average of all possible execution-times. The average-case execution-time gives a good estimate about the overall performance of an algorithm, but is of little use in real-time system design. The *best-case execution-time* (BCET) denotes the lowest possible execution-time of a function or an algorithm.

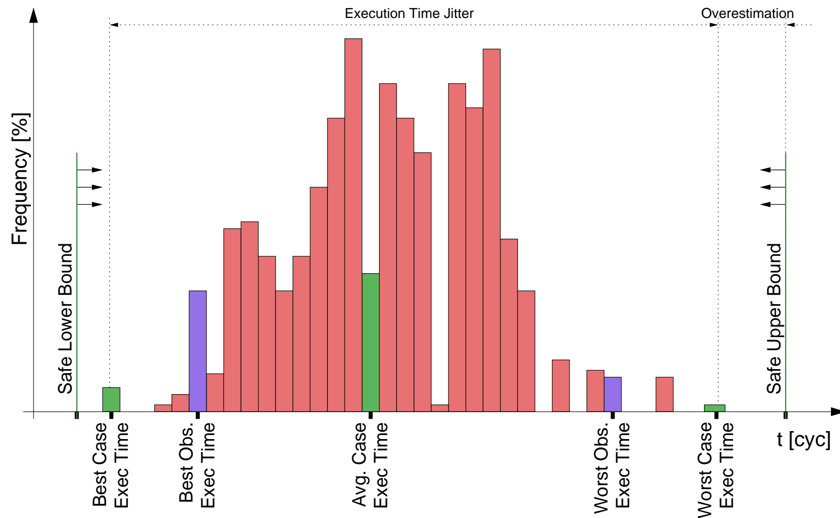


Figure 3.3: Execution Time Distribution of a Task

When execution-time measurements are performed on applications the WCET and BCET will most likely not occur, except if the user knows which input data and system state result in the WCET respectively BCET and uses this input data for measurements. It is more likely that execution-times closer to the ACET are observed. This execution-times are called *best observed execution-time* (BOET) and *worst observed execution-time* (WOET).

Finding exact values for WCET and BCET is extremely difficult therefor the goal of WCET analysis is to find a safe upper bound which is guaranteed to be higher than the WCET. In order to keep the utilization of the processor high, the *overestimation* has to be as low as possible. Overestimation is mainly introduced into the analysis of the WCET by simplifications of the hardware model or the flow facts which have to be made to keep the analysis feasible. The simplifications are bought at the price of information loss which results in overestimation but also in reduced analysis effort.

3.3 Timing analysis

Execution time analysis is used in the development of real-time and embedded systems to derive timing estimates required for schedulability analysis. Traditionally there are two methods for ET analysis: Static and Dynamic analysis. Static analysis performs analysis of applications without ever executing them, while dynamic analysis uses run-time measurements of the executed application running on the target hardware or on a simulation. In recent years the gap between both analysis methods has been filled by hybrid analysis methods which perform static analysis of the application in order to guide the measurements and to achieve better results. The following sections discuss different approaches and examine the individual steps of the analysis process.

3.3.1 Anatomy of a WCET Analysis Framework

All execution-time analysis tools use a similar strategy which is discussed in this section. Exceptions are WCET aware compilers (see section 6.1.1: *calc_wcet_167*, *TuBound* and *wcc*) which take advantage from information present in the source code and the knowledge of CFG transformations between source and object code. Generally the following steps are performed:

Syntax Analysis and CFG generation is used to generate a control-flow graph (CFG) from source or object code in order to identify basic blocks and the execution paths between them. For the majority of timing analysis methods the CFG is one of the most important starting points and the foundation for all other analysis steps. When the CFG is also used to store results from other analysis methods, i.e. the execution-time of a basic block, it is called an *annotated control-flow graph*.

Static high level analysis uses different analysis methods like abstract interpretation, value analysis, or loop and loop bound analysis which are performed without execution of the program. The results are needed to identify the entities (mostly basic blocks, sequences of BBs or paths) for which the execution-time shall be gathered in the next step and to generate input data for the execution-time acquisition (i.e. the memory locations for a given set of variables to perform cache analysis).

Acquisition of execution-times can be performed by using static low level analysis methods on an accurate hardware model, which may consider caches, pipelines, branch prediction units and even timing anomalies. As an alternative the measurement of execution-times on the target hardware or a cycle accurate simulator can be used.

WCET calculation is the last step of the analysis in which the execution-times of smaller program fragments are combined to get a WCET estimate of the whole application. This analysis step can be integrated in the static low level analysis.

The following sections briefly describe the individual steps needed for WCET analysis. Section 6.1 will give an overview of WCET analysis tools and explain how the different techniques are used in static (section 6.1.1), dynamic and measurement-based (section 6.1.2) tools.

3.3.2 Syntax Analysis and CFG extraction

Analysis tools can use different levels of program representation. Some tools operate on the source code of the application. This gives them access to additional structural and control-flow information, but due to compiler optimizations (section 2.3.2) the CFG of the source code may differ from the control-flow of the application. The

other approach is to use binary object code as basis for the ET analysis. An advantage is that the control-flow of the analysis and the executed program is exactly the same. On the other hand it is more difficult to extract the control-flow from the object code but this is only an implementation issue. It is of more importance, that some information about variable ranges, which can be extracted from the source code or are annotated by code generation tools, are lost. Of course, extracting the CGF from the object code leads to a strong dependence on a distinct hardware architecture. A different approach, which is often used by WCET aware compilers, is to combine both, source code and object code analysis (see table 6.1). This can also be achieved by generating assembler output from the compiler which contains the application's source or by embedding source line information in the generated object. The GNU C compiler *gcc* can be instructed to do so by using the `-g2` command line switch. It should be noted that some optimization options will not work correctly with all debugging options. An overview which optimization options will influence debugging code generation on the GNU gcc can be found in the gcc online documentation[GCC].

3.3.3 Static Analysis

Static analysis refers to the analysis of software applications by means of formal methods without executing them. Static analysis consists of high level or flow analysis bounding the software behavior and low level analysis and hardware modelling to bound the hardware behavior. In general static analysis has to make simplifications on the model because of the model complexity. These simplifications come at the cost of overestimation.

High-Level or Flow Analysis

The goal of static analysis is to analyze the software behavior. High level analysis is within certain parameters, like the size of datatypes, platform independent. High level analysis includes techniques like variable range analysis, abstract interpretation, loop and loop bound analysis, (infeasible) path analysis and function analysis. However it is not possible to find exact flow information since this would solve the halting problem. The following list is a short overview of methods used in static analysis.

Manual annotations are not an analysis technique but can be used to guide the analysis and supply additional information which is only available at a higher abstraction level. Many analysis tools allow manual annotations. Unfortunately each tool offers its own annotation language and there is no common standard [KKP^{+}07].

Syntactical analysis on parse trees (pattern matching) are usefull on compiler generated loops. A given compiler always uses the same pattern when generating the object code for loops. It is often possible to analyze a loop based on that pattern. For simple loops it is even possible to extract the loop bound [HS02]. On source level

there are also repeating patterns which can be observed when code generators are used.

Presburger arithmetics is a theory of the natural numbers with addition based on first-order logic. It can be extended with multiplication but only by constants. Simple loops can be modeled in Presburger arithmetics which can easily be solved to calculate loop bounds [HS02].

Data flow analysis (DFA) is used to identify dependencies of variables and control-flow decisions. To perform data flow analysis, data flow equations for each node are created. These equations are solved by following the CFG repeatedly and using the output of the last iteration until the system stabilizes and a fixpoint is reached [SP81, Bli02]. Data flow analysis can also be used to find loop bounds [CM07, dMBCS08]. Compilers use DFA to perform liveness analysis of variables.

Symbolic execution is a software analysis technique which simulates the execution of a program using symbolic values like variable names rather than actual values for input data [Cow88]. The output of symbolic execution are logical or mathematical expressions using these symbols which can be solved by means of a SAT solver. Symbolic execution can be used to identify program paths [Zha04, ZCW04, LS99a], analyze loop bounds [BB00, LS99a] and to generate test data for hybrid ET analysis [OS90].

Model checking is a formal analysis if a given finite state machine (FSM) meets a gives specification. Typically these models use a language which defines states and transitions but some model checkers can also use C as input language. Model checkers are usually used to find dead code or to check if given software requirements are hold on an examined program. However model checking can too be used to detect infeasible paths [WRKP05] and loop bounds [RPW08] as well as dead code [HJMS03].

Abstract interpretation (AI) is the mapping of the semantics of a computer program to a model based on monotonic functions over ordered sets, especially lattices. According to [CC77, CC79, CC92] an abstract interpretation is defined as a non-standard (approximated) program semantics obtained from the standard (or concrete) one by replacing the actual (concrete) domain of computation and its basic (concrete) semantic operations with an abstract domain and corresponding abstract semantic operations. AI has many uses in compiler construction and debugging. In association with high level static WCET analysis it is used for (infeasible) path analysis [GE97, GEL06] and loop bound analysis [GE97, ESG+07], as well as for value analysis and low level static analysis as discussed below.

Value analysis is a special form of abstract interpretation. It computes ranges for the values in the processor registers and local variables at every program point. The value analysis is able to determine memory locations statically [FMW97, ESG+07] which is important for a subsequent cache analysis. Value analysis is also able to detect loop bounds and infeasible paths.

Low-Level Analysis and Hardware Modelling

Cache analysis is used to analyze the memory access patterns of an application and statically predict cache hits or misses. It requires knowledge about memory access patterns. The simplest method is to assume an *always miss* cache behavior. This can lead to overestimations. However, from the static high level analysis the memory access patterns of the application are often known and therefore cache analysis can be integrated in the AI framework [AFMW96, FMW98, TF98, RM05]. The analysis uses the classifications *always hit*, *always miss* and *not classified* and defines merge functions for different cache replacement strategies to combine joining control-flow edges. Other tools include the classifications *first hit* and *first miss*. The AI-based cache analysis can be used for instruction-, data- and combined caches. Model checking as shown in [Met04] can also be used for low level cache analysis. The proposed method uses an explicit model of the cache in the processor model and works currently for small case studies but not for real-size applications.

Pipeline analysis is used to determine the effects of the pipeleine on the execution-time of the application. The pipeline analysis depends on the results of cache analysis and flow analysis since pipeline analysis has to consider possible cache misses and data hazards, which cause cache miss penalties or pipeline stalls [HWH95, HAM+99]. Both effects depend on the execution history of the processor. An abstract model is used for the representation of the pipeline state. The model defines timing dependencies between different instructions, i.e. at which pipeline stage a register has to be available for a given instruction so that no hazards occur. A similar approach is described in [FHL+01, MSR02] where the pipeline simulation is integrated in the AI framework using a formal pipeline model with defined update functions. A different approach is to use an off-line simulation of the flow of instructions through the pipelines [CP01]. The discussed pipeline analysis approaches based on AI are able to handle both in-order and out-of-order pipelines.

Branch prediction analysis is used to simulate the effects of branch predictors found in modern architectures on the application timing. It is accomplished by introducing history patterns in the AI framework [MRL02, LMR05]. The branch prediction implements a simple global 2-bit branch prediction unit as shown in figure 2.9. The solution proposed in [MRL02, LMR05] defines an update function which updates the branch history for every conditional jump taken and computes a branch prediction for each block based on the current execution history and the branch history. The experimental results shown in [LMR05] examine two global branch prediction shemes using

different hash functions for the branch history table and branch prediction scheme with a local branch prediction table. The experimental results are very close to the observations from simulation runs except for the *fft* case study because of loop bound dependencies on the inner loop from the outer loop of the fft algorithm [LMR05].

Static analysis of timing anomalies checks if timing anomalies may occur in an application and tries to get a numerical estimation of the effects. It requires the analysis of all possible instruction schedules and pipeline states within a basic block, which would increase the analysis effort significantly. The pipline dependencies are represented using an instruction graph [LRM04] which shows all dependencies between individual instructions executed on the processor. To find the longest execution-time (which does not correspond to the longest way through the execution graph) an exhaustive search over all paths through the graph would be necessary. Since the possible numbers of paths through the execution graph can be very high for large basic blocks this is often not a feasible option. A proposed solution [LRM04] is to perform a fixed-point analysis on the time intervals (instead of concrete time instances) at which the instructions enter/leave different pipeline stages.

Symbolic simulation uses an abstract processor model to simulate the execution of a program. The simulation is performed without input and the simulator has to be able to work on a partly unknown states. The method introduced by Lundqvist [LS98, LS99a, Lun02] called cycle-level symbolic execution combines flow analysis, pipeline analysis, instruction and data cache analysis as well as the final bound calculation within a single analysis phase. The results for the WCET estimates are identical with the results from the simulator using the worst-case input data, when optimistic merging functions were used, which could theoretically lead to underestimations. When using pessimistic state merging functions in the cycle-level symbolic execution Lundqvist states that an overestimation of 4-9 times the actual WCET was observed [Lun02].

3.3.4 Dynamic Timing Analysis

Dynamic execution-time analysis methods use measurements on the target hardware to generate a timing model. There are multiple measurement methods and design considerations which have to be considered. The following section discusses the most important ones, but not all design issues. A more elaborate discussion can be found in [KWRP05]. However, most dynamic execution-time analysis tools are in fact measurement-based or hybrid WCET analysis tools since they include also simple static analysis.

Rapita is a commercial vendor of a measurement-based WCET analysis tool. The online presence of RapiTime [Ltd08] gives a short (maybe a little biased) overview of WCET analysis methods:

> Measurement is the most common method of worst-case execution-time estimation in commercial use today. Measurement techniques insert profiling code into the software, recording the end-to-end execution-time of sub-systems, functions, or individual blocks of code.
> ...
> Recognizing that the best possible model of an advanced microprocessor is the microprocessor itself, hybrid approaches use online testing to measure the execution-time of short sub-paths between decision points in the code.

Even if these statements are biased they point out the very important fact that a modern processor is still the best model for a processor. Measurement-based methods use execution-time measurements on the hardware or a cycle-accurate simulation in order to avoid the complex hardware modelling process. The high effort for the implementation of a processor model with the same behavior as the processor itself is only one out of many reasons for this. The second reason is that in order to implement a processor model a detailed and complete documentation of the processor is necessary. Especially on modern processors the vendors omit some implementation details from the documentation which are crucial for understanding the correct function of a processor model, sometimes on purpose to protect their intellectual property, sometimes as a result of neglectfulness and sometimes because some details of the processor are of little interest for anyone who is not trying to create an abstract model of the processor. A short view on the errata list of a modern processor shows a second problem: Modern processors have bugs and often lots of them. These bugs may not only cause errors in the value domain, like the infamous Pentium® `FDIV` bug, but also in the temporal domain. At this point it should be noted that an error in the value domain might also cause a changed timing behavior as result of a changed control-flow.

A possible solution to this problem is shown in [FMC+07]. The proposed solution is to use the source code of the processor hardware in *VHDL* and extract a timing model directly from it. *VHDL* and *Verilog* are originally hardware specification languages but are now used for hardware design, too. For the proposed solution *Verilog* would also do fine but [FMC+07] focuses on *VHDL* presumably since it is more common in Europe. The timing model extracted from the hardware design of the processor is used to create an abstract processor model. This has to be done since the real model would be too big to handle in AI and the real *VHDL* design will only work with concrete binary input data and cannot use an abstract representation of the input data. The proposed approach seems to work fine except for the fact that there is no formal proof that the abstraction of the *VHDL* model will behave exactly the same way as the processor.

Static analysis is often judged “safer” than measurement-based analysis but due to the facts pointed out in the previous paragraph static analysis should always be complemented by run-time measurements. The next sections show two approaches for execution-time acquisition used in measurement-based systems. The first one is simulation, which needs an accurate processor model. This implies the same problems

as mentioned before for static analysis. The second approach are measurements which are conducted on the target processor.

Simulation

Execution time measurements using simulation work by executing tasks on cycle accurate processor and hardware models. The advantage is that the user has full control over the model. Which means he can, depending on the simulator, determine parameters like cache size, memory access cycles for cache hit/miss, enable or disable functional units and more. The user has full access to the processor and can even modify processor registers. The measurements have no side effect on the simulation since the user can set simulation breakpoints and read out the processor time in simulation cycles at any point.

Therefore simulators are ideal to test new approaches for WCET analysis since complex architectural features may be enabled, disabled or changed to the likings of the user. For the simulation of real processors there is always a remaining degree of uncertainty if the simulation model behaves exactly like the original processor.

In case of simulations the approach developed in [FMC+07] will work since the simulation runs on concrete input data and can be carried out by any hardware simulator like ModelSim®. One problem remaining is the availability of the processor source code in *VHDL* or *Verilog*. Even if the source code would be available, there would still be performance problems for large processors. Hardware designs of simple processors for synthesis or simulation, including the MIPS architecture which is often used for educational purposes, are freely available from the Opencores website [Pro09].

Execution Time Measurements

For modern processors which are too complex for simulations or where the source is unavailable, execution-time measurements on the hardware have to be made. Execution time measurements are defined in [KWRP05]:

Definition 11 *Execution time measurement refers to the task of determining the execution-time (or respectively the number of clock cycles) by executing a particular path through a specific code fragment of a given program on a physical hardware.*

There are multiple possible measurement techniques which can be used for this purpose. The following classification of measurement methods [Pet03, KWRP05] gives a short overview:

Simulation is not an execution-time measurement by definition, but included here for completeness. Simulation have been discussed in the previous section.

Light-Weight Software Monitoring uses an internal counter (i.e. cycle counter of the CPU) as time basis and relies on software instrumentations placed in the executed application. A possible instrumentation technique is to read the time basis and write the result to a specified memory location or variable. Light-weight software monitoring has a neglectable impact on execution-time and instrumentations can remain in place after the measurements are finished so the measured and real execution-times are the same. Light-weight software monitoring does not enforce a known state on the hardware and should therefore only be used on simple architectures [KWRP05]. The execution-time measurements in this thesis use light-weight software instrumentations.

Heavy-Weight Software Monitoring has to be used on complex architectures where the execution-time may be strongly affected by the processor state. In order to achieve valid and reproduceable results the state of the hardware must be known, which implies that it is eventually enforced, at the beginning of each measurement. This hardware state modification may result in increased execution-time. On architectures where timing anomalies may occur the instrumentations have to remain in place too, otherwise they should be removed [Pet03]. Like light-weight software monitoring this technique relies on an internal counter.

Hardware Supported Software Monitoring places instrumentation code in the program producing events which are observable from the outside of the measured systems. Instrumentation points may toggle an I/O-pin or write an ID to a group of I/O-pins. The change on the pins can be monitored using an oscilloscope or a logic analyzer. It should be taken into account that changes from *high* to *low* can have a different timing than changes from *low* to *high*. Since the approach has little effect on the execution-time the instrumentations can remain in place after the measurements have been completed. The disadvantage of this method is that expensive external hardware is needed.

Hardware Monitoring can be implemented either by monitoring the bus traffic (which will only work for architectures without internal D-cache) and analyzing the instruction flow on the bus or by using the debugging interface present on many embedded chips. Using the debugging interface requires additional hardware which comes of prices from less than 100€ to more than 10,000€ for trace probes with real-time and in-circuit emulation capabilities. For hardware monitoring no alterations on the software have to be made. The disadvantage is that cheap solutions are rather slow and fast solutions are rather expensive.

As many measurement techniques have an impact on the execution-time, it is favorable to increase the size of the measured fragments as much as possible. However, when the size of the code fragments exceeds the size of the basic blocks, input data dependent control-flow occurs during the measurements. Since measurements are

only meaningful when the executed path is known, the path has to be enforced for the measurements. There are two possibilities of enforcing the execution of a specific control-flow path:

Code Modifications in order to replace conditional jumps by unconditional jumps or NOPs to force a specific path [PF99]. This can affect the timing behaviour in many ways. First, the timing of NOP, unconditional and conditional jumps are likely different. Second, on architectures with speculative execution the branch prediction is altered. Therefore, code modifications to enforce a given control-flow are not favorable.

Test Data can be used to enforce a given path. Since all conditional jumps of feasible paths are either test data dependent or depend on static data (i.e. loop counters) it is possible to generate test data which enforce the execution of a given path. To measure a set of given paths test data for each path have to be generated and the execution-time of the application running with the generated test data is measured. Since no alterations of the application code need to be made this solution gives more reliable results than code modifications.

A list of other important design considerations [KWRP05] includes multiple aspects which have to be considered for the selection of the measurement method and data acquisition strategy, especially for small systems with limited resources (CPU, memory and I/O).

Counter register location: Is a processor internal counter used or is the counter located within an external time source?

Hardware or software instrumentation: Is hardware or software instrumentation used?

Interface data: Which data needs to be exchanged between the host and the target system? Are complete traces or only execution-times to be stored?

Required devices: Is there special hardware needed that is dedicated to execution-time measurements (e.g., logic analyzer)?

Persistence of code instrumentations: Do the instrumentations remain in the code after the run-time measurements (persistent instrumentation) or are the instrumentations removed after the measurements (non-persistent instrumentation)?

Recording location: Where is the observed execution-time information stored? Is it stored in the same computer system (internal) or on an external device (external storage location)?

How often has the code to be instrumented/recompiled: Has the code to be recompiled and downloaded to the target for each instrumentation or are all instrumentations placed within one executable simultaneously?

Control flow manipulation: When using active instrumentation, the control-flow

is enforced by code modifications. On the other hand, passive instrumentation refers to instrumentation gathering dynamic execution information.

Input data generation location: Are the input data generated on the host or on the target?

Representation level: At which representation level are code instrumentations performed (C-code, assembly code or object code level)?

Number of measurement runs: How many measurement runs can be performed with one modified execution binary (=one code image that contains instrumented code)?

Resource consumption: How much resources, like static memory, data memory, bandwidth, output ports (interface lines), run-time cycles are required by the selected method?

Installation and usage effort: Is human installation required (e.g., plug a special socket on a chip or add additional wires)?

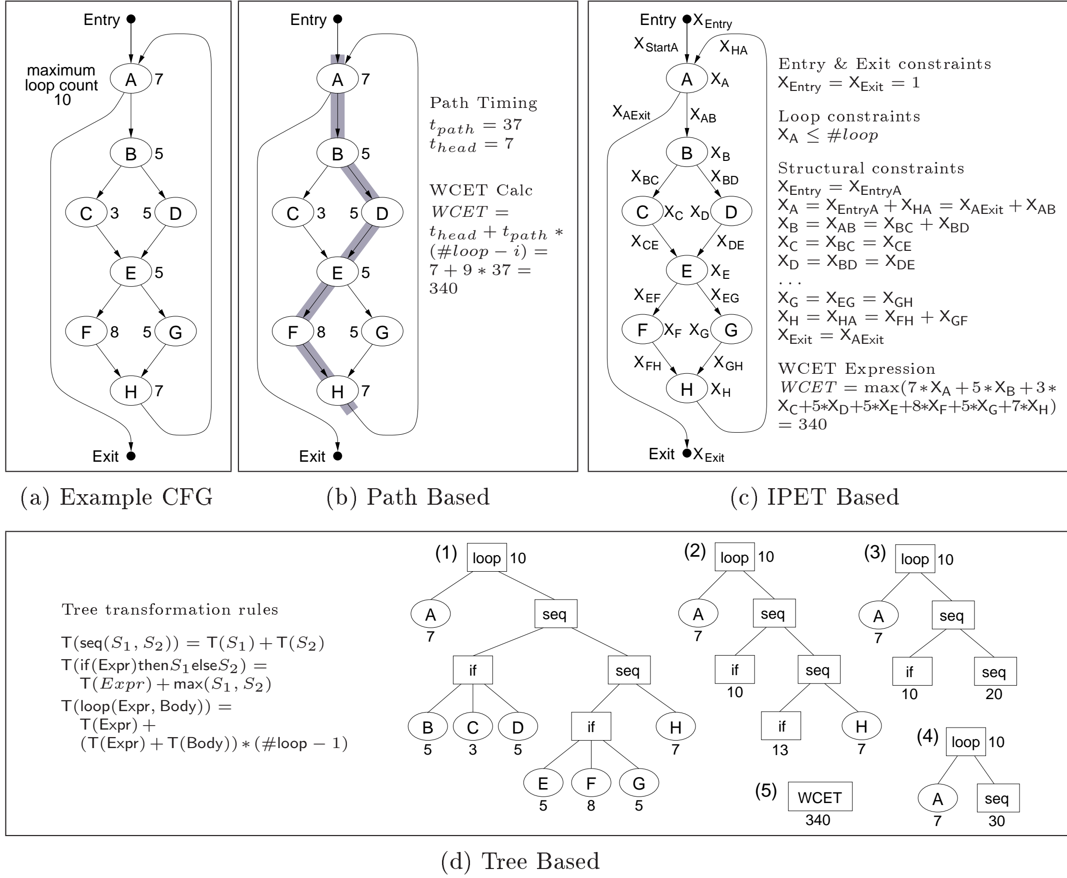
Each of the listed design considerations has to be given thourough consideration. The choice of the best measurement solution can vary-based on different application properties even on the same target hardware.

3.3.5 WCET Calculation

The final WCET calculation step can be done using three different approaches which are shown in figure 3.4. The *path-based* method shown in subfigure b uses explicitely specified execution paths, while *implicit path enumeration technique* (IPET) illustrated in subfigure c uses implicit path information generated by system of equations to model the control-flow and *tree-based* often called *syntax-based* methods displayed in subfigure d are oriented at the program structure. The discussed WCET calculation techniques are performed by static or hybrid analysis tools. Traditional measurement-based tools without any form of static analysis cannot perform these WCET calculations because of the lack of flow information and rather use end-to-end measurements.

Tree or Syntax Based WCET Calculation

Tree-based WCET calculation techniques use static analysis or measurements to gather the execution-times of basic blocks and combine the WCET of individual basic blocks to an estimation of the global WCET using simple rules-based on the structure of the application source [PS90]. On the left side of figure 3.4d the three transformation rules which are applied in order to calculate the WCET are shown. They are very simple and intuitively to understand which is one of the advantages of tree-based timing schemata. Figure 3.4d shows how the rules are iteratively applied until the final WCET is calculated.



(a) Example CFG (b) Path Based (c) IPET Based

(d) Tree Based

Figure 3.4: Overview of different WCET calculation methods [Erm03]

The concepts of scopes and markers is introduced in [PK89, PS93] in order to tighten the WCET estimate. A scope is defined in [PK89] as a part of a program's instruction code, limited by a special scope language construct, that is embedded into the syntax of a programming language. Markers are defined as special marks located within a scope. They specify the maximal number the marked positions in the program may be passed by the program flow between entering and leaving the scope. An arbitrary number of markers may be placed within a scope. The concept can dramatically improve the WCET estimates because it can be used to introduce information that is only available in a higher representation level of the application into timing analysis.

Another extension made to the syntax or tree-based approach is the ability to extract the syntax graph which is the basis for WCET calculation from the (non-stripped) object code of applications and to use execution-time profiles instead of simple execution-times for the WCET calculation [BCP02, BCP03]. The generated results do not only give a WCET estimation, but a full profile of execution-times and the probability of their occurrence.

Path Based WCET Calculation

In path-based WCET calculation, the upper bound for a task is determined by computing WCET bounds for different paths in the program, searching explicitly for the path with the longest execution-time [HAM+99, HW99]. The important characteristics of this approach is the explicit representation of the path. The WCET analysis method developed by Healy [HAM+99, HW99] includes the static analysis of pipeline timing and cache behavior.

Stappert et al. separate the flow analysis from the hardware analysis [SA00]. This makes the flow analysis independent from the hardware and helps to reduce the complexity of the models. A further improvement to this approach makes it able to handle loops and flow facts, which cause problems with the path-based method because of the exponential growth of the number of paths. Eventually heuristic search methods [SEE01] are required to find the longest path. Infeasible paths which present a problem for path-based WCET calculation, since their infeasibility has to be proofen, can be detected by abstract execution [GEL06] or model checking [WRKP05].

Implicit Path Enumeration Technique

Because the Implicit Path Enumeration Technique is-based on the CFG it is sometimes referred to as graph-based WCET calculation. The basic idea behind IPET is to combine the program flow information and the maximum execution-time of basic blocks into a set of arithmetic constraints. The program flow starts at the entry node of the program and leaves it through the exit node. The flow between nodes is represented by X_{AB} where A represents the node which has been executed and B which will be executed next. Similar X_A represents how often basic block A is executed. For a single node j the following equation holds:

$$X_j = \sum_{\forall i} X_{ij} = \sum_{\forall j} X_{jk} \qquad \forall i, j, k : X_{ij}, X_{jk} \in \text{CFG} \tag{3.1}$$

This equation can be created for each node of the CFG forming a set of structural constraints. Additionally a constraint for each loop can be given assuming basic block i is the loop head of a loop bounded by a maximum of Bound_i iterations:

$$X_i \leq \text{Bound}_i \tag{3.2}$$

The equations for the entry and exit nodes are :

$$X_{\text{ENTRY}} = X_{\text{EXIT}} = 1 \tag{3.3}$$

Equation 3.1 to 3.3 bound the number of executions of individual basic blocks while allowing all possible end-to-end control-flow paths through the program. The

execution-time of a single basic block within the WCET path is given by the frequency of its execution X_i times its execution-time bound C_i. To the execution-time for the whole program the sum of all execution-times in all basic blocks is calculated $\sum_i X_i * C_i$. Finally an upper bound for the ET can be calculated using equation 3.4.

$$\text{WCET} \leq \max\left(\sum_{\forall i} X_i * C_i\right) \qquad \forall i : X_i \in \text{CFG} \tag{3.4}$$

IPET was first proposed by Malik [LM95, LM97, PS97] and adopted and further developed and improved by many others [Eng02, The02a, The02b, Erm03]. IPET can handle different types of flow information. Usually IPET is applied to a whole program. The bound calculation is performed using an ILP solver like *lp_solve* [Ber97] or the GNU Linear Programming Kit (GLPK) [FSF09] which is used in the presented approach. When flow facts are also integrated in the IPET model constraint programming (CP) techniques are used to solve the equation system, which grows very fast in that case.

3.4 The Measurement-Based or Hybrid Timing Analysis Approach

This section gives a short introduction about the theoretical ideas and concepts behind measurement-based WCET analysis. General problems which occur during the hybrid WCET analysis are discussed and it is explained how these problems are concretely resolved by the WCET analysis tool kit presented in this work. Technical details and implementation details are presented in chapter 4.

Measurement-based execution-time analysis frameworks do not need a complex abstract processor model since they use the hardware to perform execution-time measurements. It seems simple, but the most important question when performing measurements is: "What do I want to measure?". In the industrial daily routine often end-to-end measurements are performed using test data from normal operations or user generated test data. However, the path being measured is unknown and it is completely coincidental if the WCET path is ever executed during these measurements.

To solve this problem the measurements have to be guided to find the path with the longest execution-time. The following sections describe how different paths can be identified for measurement and how test data is generated to execute these paths. Since the number of end-to-end paths grows rapidly with the application size, it becomes soon infeasible to perform measurements of all end-to-end paths. The proposed solution to split the application under test into smaller parts, called *program segments* (PS), is also discussed in the following sections.

3.4.1 Assumptions and Prerequisites

For the further elaborations about the hybrid WCET analysis approach the following assumptions about the usage of the C language features, the compiler and the target hardware are made:

Programs have to be structured. Loops are created with *for*, *while* or *do ... while* constructs; *goto* and *<label>:* are not used for the implementation of loops.

Type qualifiers and storage class modifiers are used correctly. The sample code from an industrial application which is used as a test case uses declarations like `extern volatile *someVariable;` to declare constant values stored in the application ROM. This causes all ROM constants to be treated like input parameters for the simulation and increases the state space for the model checker enormously. The WCET analysis prototype performs a very simple data flow analysis which has two categories for variables - *const* and *variable*. The analysis is very simple but when used in combination with loop analysis it can significantly improve both, the analysis effort as well as the analysis result. Since variables are categorized by their access patterns (LHS or RHS expression) the *const* modifier is only needed for extern variables which are initialized outside of the current source module.

Programming according to the MISRA and DO-192 guidelines, which does for instance not allow dynamic memory. However, conforming to these guidelines should anyway be imperative for safety-critical systems.

The compiler does not change the control-flow. This restriction is necessary because the analysis tool works on source code and makes assumptions about different control-flow paths which are used to perform measurements of the same paths on the target hardware. Therefore, the CFG of the source code has to match the CGF of the object code. It is not always completely clear which compiler options modify the control-flow on a given compiler. Kirner [Kir03] examines different optimization techniques and their effect on control-flow. This requirement is certainly a disadvantage since it forbids the use of several compiler optimization techniques.

The hardware does not suffer from timing anomalies. This is important for the separate measurement of program segments. On simple architectures like the *ARM*® instrumentation points only require constant time to execute, which can be subtracted from the measurement result. Additionally, space in application object code and memory to store the results is required. On more complicated architectures the measurements also have an influence on the cache behavior. Therefore heavy-weight software instrumentations have to be used on these architectures, which means that the cache has to be flushed at the beginning of each measurement to resemble the worst-case (under the assumption that an empty cache is the worst case). On architectures with timing anomalies an empty cache is not necessarily the worst-case and therefore the cache flushes have to remain in the code after measurements which reduces the application performance significantly. Therefore the proposed WCET analysis method should not be used on these architectures until a solution for this problem is found.

Figure 3.5: Basic Hybrid WCET Analysis Approach

3.4.2 Basic Idea - Guiding Measurements by Static Analysis

After the general requirements have been defined in the previous section this section gives an overview of the proposed hybrid WCET analysis approach. Figure 3.5 shows the basic architecture of a hybrid WCET analysis tool.

The input for the analysis is the application source code in C. Based on this source code static analysis is performed. In the basic approach which is described here this is primarily the identification of control-flow altering variables and control-flow paths. Based on the identified variables and the control-flow, test data is generated to cover all feasible control-flow paths. At the same time the compiler generates an instrumented executeable of the application, which is used together with the generated test data to conduct execution-time measurements. The resulting execution-times are combined in the WCET calculation step to calculate a safe WCET bound. Since the generated test data covers all paths the WCET path is certainly executed.

Figure 3.5 illustrates the individual steps of hybrid WCET analysis. A portion of the analysis is carried out using static methods, the other portion is performed using execution-time measurements. The individual steps are described briefly below:

1. The first step is static analysis. During this step all control-flow paths through the application are analyzed as well as loop bounds, functions calls and control-

flow information which is hidden in logic expressions and cannot be seen from the program structure. However the number of end-to-end paths is very high, which is the first essential problem. The proposed solution which is anticipated in figure 3.5 is to split the application into smaller fragments called program segments PS (see next item).

2. The second step, called CFG partitioning, is combined with the first step. Its task is to reduce the number of end-to-end paths by splitting the applications into smaller program segments (PS). A description, why the number of measurements can be reduced using this approach, can be found in the next section.
3. The third step is test data generation. In order to obtain safe WCET results all paths within a PS have to be covered. However, there are often infeasible paths which cannot be executed. These paths need to be identified. The test data generation and the identification of infeasible paths is the second essential problem to be solved.
4. After the test data are been generated all paths, except infeasible paths, can be measured. In other words, the measurements will surely cover the path with the longest execution-time.
5. The fifthth and last step is to combine the measured execution-times from the individual segments into a single estimate of the WCET by using the structural information gained in steps 1 and 2.

This short description of the hybrid WCET analysis approach points out two important problems:

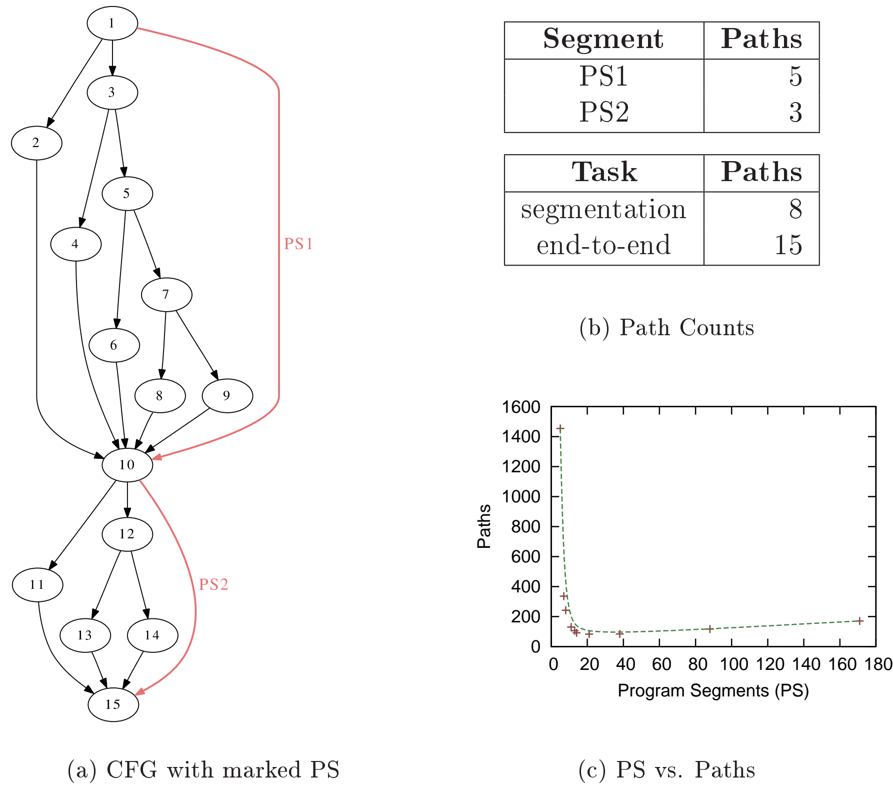
Feasibility: The feasibility of this method depends on the number of end-to-end control-flow paths. The proposed solution works for small case studies but real applications often have a total of 10^{10} to 10^{100} control-flow paths. If the analysis of a single path could be performed in $100\,ms$, including test data generation and measurement, the analysis of 10^{10} paths would still take 31.7 years.

Test data generation can clearly only be performed for feasible paths. The analysis framework has to identify infeasible paths and create test data for all other paths.

The next sections will discuss how to solve these problems. Other problems occur when function calls and loops are to be analyzed. Possible solutions for the encountered problems are discussed in the next sections, too. A description of the technical realization for the proposed solution can be found in chapter 4.

3.4.3 Partitioning or Segmentation of the CFG

A solution to the problem of the high number of paths is to divide the control-flow graph in smaller parts called program segments (PS) and only analyze the subpaths of these segments. This reduces the total number of paths.



Segment	Paths
PS1	5
PS2	3

Task	Paths
segmentation	8
end-to-end	15

(a) CFG with marked PS

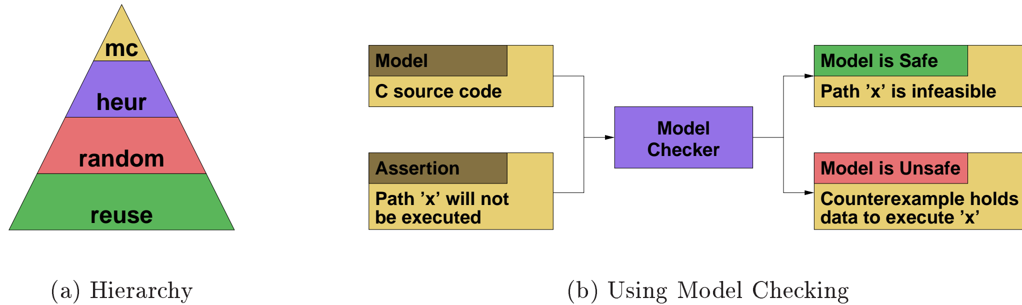
(b) Path Counts

(c) PS vs. Paths

Figure 3.6: Concept of CFG Partitioning

Figure 3.6 explains how partitioning works. A simple CFG is shown in subfigure a. Subfigure b shows a table with the path counts for subfigure a. In the whole task there are 15 end-to-end paths. If the CFG is broken up into two program segments which are marked as $PS1$ and $PS2$ and contain 5 respectively 3 paths the number of total paths drops to 8. This comes from the fact that both PS are evaluated independently and when analyzing $PS1$ the control-flow in $PS2$ is not considered. The separate analysis reduces the multiplication of path counts for consecutive segments to an addition. The larger the programs get, the larger the reduction of path counts. This technique makes even programs with more than 10^{100} end-to-end paths analyzeable. Subfigure c, which is taken from [WKRP08], shows the relation between the number of paths which have to be analyzed and the number of program segments.

As shown in section 3.3.4, instrumentation for execution-time measurements can change the behavior of the application. In order to get accurate results measurements should be taken from larger portions of code. The analysis framework performs the segmentation automatically using a parameterizable path bound. The path bound denotes how many control-flow paths may be contained within a PS before it is broken up to smaller segments. Wenzel [Wen06] has examined the relation between



(a) Hierarchy (b) Using Model Checking

Figure 3.7: Test Data Generation

the number of PS and control-flow paths, and shown that a path bound of around 100 results in a good compromise between the number of paths which have to be analyzed and the accuracy of the analysis for most applications. Therefore a path bound of 100 has been selected for the experiments is chapter 5.

3.4.4 Test data generation

The test data generation has to be completely automatic. It is not adequate for an analysis tool to rely on user generated or, even worse, random input patterns. However, random patterns are used in the proposed approach, too, as part of a data generation process which combines random generated test data, heuristics*, and model checking. The implementation details of test data generation are described in section 4.13. The test data is stored and reused for consecutive runs of the analysis tool. Figure 3.7a shows the applied hierarchy of test data generation.

Model checking applies formal methods to finite state machines (FSM) and checks a set of given assertions if they hold within the model or not. Model checking is complete, which means if a model violates a given property this is guaranteed to be detected by the model checker. The model is extracted from the C source code by the model checker. The assertion has to be something that can be negated by the model checker like “Path 'x' can never be executed”. As figure 3.7b [RWSP07] shows, the model and the assertion are used as input for the model checker. When the model checker cannot disprove the assertion, the given path can never be executed and is infeasible. Therefore no test data is required for measurement. If the model checker can identify a case, where the given assertion is wrong, which means that the given path can be executed, the counter example yields the test data required to execute the analyzed path. The technical realization of this test data generation process is described in section 4.13.

*Heuristic data generation was planned to reduce the number of paths which have to be model checked but never implemented since during the experiments the test data generation worked fast enough without it.

3.4.5 Function Calls

There are two possible ways to resolve function calls: First the function can be analyzed separately. During the analysis of the calling function the function call is replaced by a basic block and the WCET bound of the function is assigned as execution-time to the basic block. This solution is referred to as *black-boxing*. Second the function can be inlined similar to C++ inline functions, an approach which is referred to as *inlining*. Inlining leads to duplication of code, which is a drawback especially in small systems with limited memory resources. Therefore the inlining is only done during the analysis and test data generation. The code is not inlined in the target code when measurements are performed.

Black boxing can lead to an overestimation of the execution-time for functions which are normally called with a limited parameter set. Especially functions which are called with constant parameters may be unbounded without their usual parameter set.

Inlining takes the context from which the function is called into account and analyzes the control-flow inside the loop considering the parameters given by the caller. The parameters may be limited in their values by the caller or even be constants which is especially important when the called functions contain loops.

Experiments that compare the WCET estimations using black-boxing and inlining can be found in section 5.4. The reduction of overestimation using inlining is only small for most applications.

3.4.6 Simplified Data Flow Analysis

Data flow analysis examines the dependencies amongst variables and is a common analysis performed in compiler construction. The presented approach uses a simplified forward data flow analysis, which is practically a const analysis without value propagation. The analysis process is described in section 4.7. It can only categorize if a variable is *const* or *var* at a given basic block, depending on the usage of the variable in assignment expressions. Side effect operators effecting the variable are analyzed, too. In the context of this analysis *const* denotes that the value of a given variable is not input data dependent. For instance the loop iterator variable inside a loop is treated as *const* if all operations on it, which are the initial assignment, the change of its value for each iteration and the checking of the loop exit condition, depend only on constant values. Therefore the values *const* and *var* denote the combination of value and control-flow dependency rather than the actual value alone.

Loops require special handling. Iterations are performed during the analysis until a fixpoint is reached. After the simplified DFA algorithm finishes it is known for every basic block whether a variable holds a constant or a variable value.

3.4.7 Loops

A generic algorithm for loop timing analysis is to find the path with the biggest execution-time for each iteration and to calculate the sum of these execution-times over all iterations. It has already been described in section 3.3.3 how the loop bound can be determined by abstract interpretation, symbolic execution, Presburger arithmetics, pattern recognition or model checking.

The proposed solution to analyze loops is therefore to generate test data for each iteration to cover all paths within the loop. This solution requires $N * |\Pi_{\mathrm{Loop}}|$ paths to be analyzed where N is the loop bound and $|\Pi_{\mathrm{Loop}}|$ is the number of paths contained in the loop. As the analysis of loops in section 5.3 shows, the effort for this approach can be quite high.

The simplified data flow analysis can be used to categorize loops. The simplest loop type is the single-path constant-iteration (*SP/CI*) loop which can be recognized because only *const*-variables are involved in the control-flow decisions of the loop header and body. The execution-time of this loop can be measured without even generating any test data since the control-flow through the loop is always the same.

Single-path variable-iteration (*SP/VI*) loops contain no data dependent control-flow in the loop body, but may contain data dependencies in the iteration condition in the loop header. For these loops test data is generated so that the loop is executed with the maximum possible iteration limit. Like for SP/CI the loop can be measured without instrumentations in the loop body.

Multiple-path variable-iteration (*MP/VI*) loops require the generic loop handling approach discussed before. This requires an instrumentation point inside the loop body. This is not a big issue for the *ARM*® architecture but for architectures with cache this can cause serious execution-time penalties and overestimations.

3.4.8 WCET Calculation Step

The WCET calculation step is easy once the execution-times for the individual program segments have been determined. To calculate a WCET bound a longest path search algorithm applied on the directed graph of program segments is sufficient. In the presented WCET analysis framework the WCET calculation is implemented using IPET. Using IPET and creating an ILP for the final WCET calculation has the advantage that flow facts, like the mutual exclusion of paths, can be integrated in the process. An interesting possibility would be to use n paths with the longest execution-time from each segment and check for mutual exclusions between them using the model checker. The result could then be considered within the IPET model [WKRP08].

3.4.9 Differences to MoDECS V2

The main goal of the new prototype is to show that the analysis of loops and function calls is possible with the hybrid WCET analysis approach developed during

the *MoDECS* project. Further the method has been extended to support multiple compilation units with correct visibility of variables and functions. Additionally the analysis of paths caused by C short circuiting in logical AND (&&) and OR (||) expressions and the conditional expression operator (?:) as described in section 2.1.2 has also been integrated.

The internal data structures of the V2 implementation could not be adapted with reasonable effort for the needs resulting from the new requirements, which made a complete rewrite necessary. Some of the basic algorithms explained in [Wen06] are still used but have also been adopted and extended to support the new data structures.

3.5 Summary

In this chapter the notion of the worst-case execution-time and their importance for real-time systems engineering has been explained. The concepts of response time, event-triggered and time-triggered real-time systems have been shown. The differences between static and measurement and hybrid WCET analysis have been shown. The keystones of high and low level static analysis have been pointed out as well as different measurement techniques for measurement-based and hybrid WCET analysis frameworks. Last the basic ideas for the hybrid WCET analysis approach, which is used in the presented prototype have been introduced.

Chapter 4

Execution-Time Analysis Framework

This chapter discusses the implementation of the WCET analysis framework, starting with the development environment. The individual steps performed by the hybrid WCET analysis framework, their implementation and emerging problems are also discussed. The individual stepsare illustrated using the example code shown in figure 4.1a. Control-flow in expressions aka. expression short-circuiting is explained with its own example since it would increase the complexity of the example application considerably.

4.1 Development Environment

The WCET analysis tool was developed under Debian GNU/Linux using XEmacs and KDevelop as programming environment. Multiple free libraries were used during the development to perform common tasks like command line parsing, solving lp problems and pattern matching (see appendix B). The analysis tool is written nearly completely in C++ except for the code which runs on the target. The target code, both manually written and generated, uses ANSI-C.

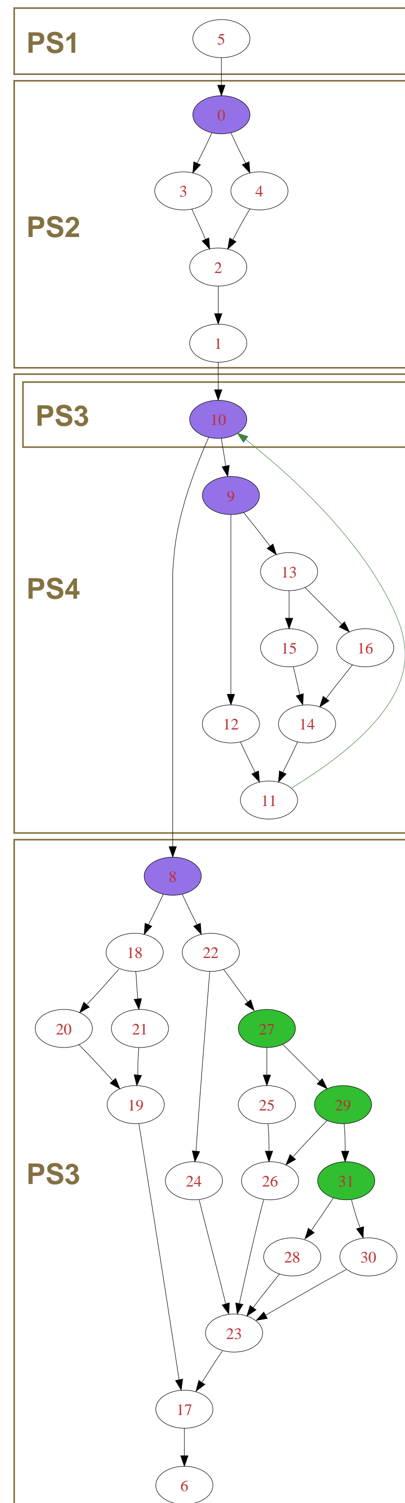
The development was done on a PC equipped with an Intel® Core™2 Duo processor running at $2\,GHz$ and with a main memory of $2\,GiB$ RAM which was funded from the FFG grant.

4.2 Example Application

Figure 4.1a shows the example which is used to explain the function and implementation of the analysis framework in this chapter. It includes a function call and a loop. For simplicity reasons there are no logical expressions with possible short-circuits present in the example. There is a separate example with logic expressions at the end of this chapter in section 4.6.

```
 1          int a, b, c, j;
 2
 3          int equal (int _a, int _b)
 4          {
 5            int rv;
 6    0       if (a==b)
 7    3         rv == 1;
 8            else
 9    4         rv == 0;
10    2       return rv;
11    1     }
12
13          int my_main()
14          {
15            int i;
16            int k = j-j+1;
17
18    5       c = equal(a, b);
19
20   10       for (i=1; i<=5; ++i) {
21    9         if (i == 1) {
22   12           a++;
23   13         } else if (i % 2) {
24   15           i += k;
25              } else {
26   16           b++;
27   14         }
28   11       }
29
30    8       if (a) {
31   18         if (b) {
32   20           a=0;
33              } else {
34   21           b=1;
35   19         }
36            } else {
37   22         switch (b) {
38                case 1:
39   24             a=1;
40                  break;
41                case 2:
42   25             b=1;
43                case 3:
44   26             c=a;
45                  break;
46                case 4:
47   28             break;
48                default:
49   30                 a=b;
50   23         }
51   17       }
52    6     }
```

(a) Listing of example.c



(b) Segmented Control Flow Graph (CFG)

Figure 4.1: Running Example for Chapter 4

4.3 Input Parsing

The lexical analysis of the input is based on flex [Pro08, App99] and uses a lex specification file based on the ANSI-C draft grammar published by Jeff Lee in 1985 and updated in 1998 to conform to ANSI X3.159-1989 which is the C89 standard. The grammar used for bison, which is the GNU version of yacc [Joh75, App99], is based on the same source. Both files are available from `ftp://ftp.uu.net/usenet/net.sources/ansi.c.grammar.Z`. The grammer had to be extended to support typedefs. Typedefs are by the C language definition a storage class for variables. However, they do not declare variables but types instead. These types have to be recognized by the lexer, even during the definition of the new data type and cannot be detected with a LALR(1) parser. An example that cannot be processed by a LALR(1) parser is

```
typedef struct { char data; lptr* next; } lptr;
```

where the data type lptr, a type representing a single linked list, is used in its own definition.

Additionally all non-ANSI extensions like `__asm__`, `__cdecl`, `near`, `far`, `__inline`, etc. are ignored by the lexical analyzers and therefore they are no longer present in the generated code. Comments are also ignored. Include-files and definitions are processed by calling the GNU *cpp* preprocessor.

4.4 Code Generation

In order to perform measurements or model checking, code has to be generated. The code generator uses the syntax tree and reproduces the input code. Since comments and empty lines as well as linebreaks are not generated, the generated code has a different layout than the input code. The code generator tries to make the code as readable as possible. Additionally the generated code is processed by the GNU *indent* tool. Some constructs are already modified by the parser. Commands which use expression statements like `if (x) ++x;` are altered by the parser to compound statements like `if (x) { ++x; }`. This is necessary so that instrumentation code can be inserted in the *if*-part of the conditional statement without altering the control-flow and semantics. In order to access static variables the names of static variables are altered and moved from the function scope or one of its sub-scopes to the global scope which is also done by the code generator. Additionally instrumentation code can be inserted on demand. To create target code, a stub function at the end of the application is created, which assigns the generated test data to global or static variables. This process is described in section 4.15.

4.5 CFG Generation

The control-flow graph is created from the syntax tree which is generated by the parser. The CFG generation uses a recursive approach. When the CFG for a specific

function is generated, the start and the exit nodes are allocated. The *build_ cfg_ rec* function is called with a syntax tree node (type *stn*) which represents the function declaration and the CFG node (type *cfg*) which represents the function start as argument. In the following CFG refers to the control-flow graph while cfg refers to the *cfg* data structure which represents a single CFG node. The cfg data structure includes pointers to the left and to the right successor node. The $\emptyset$ denotes the value of pointers which do not reference to anything similar to the NULL pointer in C.

Figure 4.2 shows a simplified CFG building algorithm which accepts compound statements, expression statements, conditional statements, function calls and *return* statements. The control-flow in expression statements is not considered and the conditional statements are limited to the form "*if (...) A else B*" where the else part is mandatory. This is only a small subset of C but enough to explain the CFG building algorithm.

Before *build_ cfg_ rec* is called, two cfg nodes are allocated, for the entry and exit points of the function. By default all pointers are initialized with $\emptyset$ in the constructor of cfg, so the address of the exit point is assigned to the left successor member variable *left_ child*.

When *build_ cfg_ rec* is called with *stn_ node* pointing to a simple expression statement, the statement is appended to the statement list of the entry node and a pointer to the entry node is returned.

If *stn_ node* is pointing to a conditional expression, then three cfg nodes are created for the if-side, the else-side and for the control-flow merge. The successor of the merge node is set to the successor of the start node. The left and right successors of the start node are set to the left and the right node. The left successor of the left and right node are set to the merge node. This concludes the construction of the control-flow structure of the if statement. By calling *build_ cfg_ rec* recursively the statements on the left and on the right control-flow branches are filled with the statements from the left and the right syntax tree branch. After this is finished the merge point is returned to the calling *build_ cfg_ rec* function where the statements following the if statements can be appended.

For compound statements a *build_ cfg_ rec* is called for each statement. The return value is stored in *act* and used as input value for the next call to *build_ cfg_ rec* and as return value for the *build_ cfg_ rec* function. Through this method the next statements will be appended at the merge point in case there is a conditional expression encountered. It is important for statements like *return* that the exit point of a function or merge points are never changed.

For functions two new nodes are produced and integrated in the control-flow. Both nodes are marked specially to enforce segment boundaries. For inlined functions *build_ cfg_ rec* is called with the first cfg node and the definition of the function in the syntax tree. The exit point of the function is stored on a global stack. When a return is encountered, the address held in the top value of this stack is used as target. When the function processing is finished the exit point is removed from the stack. For black-boxed functions the function is called in the PS defined by the two newly

```
build_cfg (stn* function)
    stn *start=new(stn), *end=new(stn)
    start.left_child = end
    build_cfg_rec (function, start)
    return

cfg* build_cfg_rec (stn* stn_node, cfg* start)
    // add expressions directly to the actual node
    if typeof(stn_node) == 'expression_statement'
        start->add_statement(stn_node)
        return start
    // build CFG graph skeleton for if statement
    // fill left and right side, then add next
    // statement to CFG merge point
    if typeof(stn_node) == 'selection_statement'
        stn *left=new(stn), *right=new(stn), *merge=new(stn)
        left->left_child = merge
        right->left_child = merge
        merge->left_child = start.left_child
        start->left_child = left
        start->right_child = right
        build_cfg_rec (stn_node->left, left)
        build_cfg_rec (stn_node->right, right)
        return merge
    // add each statement to compound statement
    // consider changes of the actual CFG node
    if typeof(stn_node) == 'compound_statement'
        stn *act=start
        for each child in stn_node.get_childs do
            act = build_cfg_rec (act, child)
        return act
    // build CFG graph skeleton for function, perform function
    // call, manage return address stack, mark nodes for PS
    if typeof(stn_node) == 'function_call'
        stn *fu_entry=new(stn), *fu_exit=new(stn)
        fu_exit->left_child = start->left_child
        start->left_child = fu_entry
        fu_entry->left_child = fu_exit
        global_return_stack.push(fu_exit)
        build_cfg_rec (stn_node.function, fu_entry)
        global_return_stack.pop()
        fu_entry.mark_for_segmentation()
        fu_exit.mark_for_segmentation()
        return fu_exit
    // jump to top element of return address stack
    if typeof(stn_node) == 'return'
        start.left_child = global_return_stack.top
        return $\emptyset$
```

Figure 4.2: Basic CFG building algorithm

created nodes and the WCET of the PS is set to the WCET of the called function which is specified on the command line or in the parameter file.

The real implementation is more sophisticated since it has to check all statements for control-flow paths, which was omitted in the basic example. Also stacks have to be used to store the exit points of loops and case constructs, but the principal method of performing a top-to-bottom traversal of the syntax tree and inserting it between a floating start point and a fixed end point in the control-flow graph remains the same. The CFG generation supports unstructured constructs like *goto*, *exit* and *return* but unstructured programs can prevent segmentation or sometimes crash the prototype if it tries to perform segmentation. It is safe to use these language constructs in small functions which are not subject to segmentation and therefore the analysis tool will print a warning, but it will still try to perform timing analysis. However, unstructured programming should not be used since unstructured programs are error prone and hard to maintain.

Based on an extended version of this algorithm the CFG in figure 4.1b has been created. The application used to create this CFG was the application example in figure 4.1a. The numbering of basic blocks corresponds to the order in which the CFG nodes are created by the described algorithm. The marked nodes 0, 8, 9 and 10 in figure 4.1b are requested segment borders from the CFG generation algorithm. The second set of marked nodes, node 27, 29 and 31 are nodes needed for the construction of a case statement. Since a CFG node has only a left and a right child a case statement has to be modelled using helper nodes which do not correspond to the actual control-flow. The marked nodes are ignored and not used any further except for the transversal of the CFG. The nodes are used neither for the dominator nor for the postdominator tree nor for the dtree. Since these nodes are only virtual nodes instrumentations cannot be placed in them too.

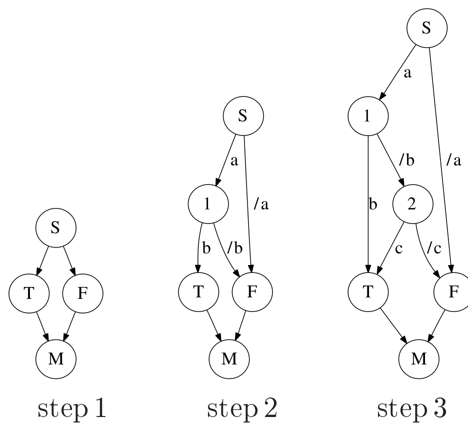
4.6 Expression Paths

Paths inside expressions are an important feature of C. Expressions with control-flow paths are not included within the application example to keep the example and the data structures generated from the example within reasonable size. The minimalistic example shown in figure 4.3 will be used to outline the expression paths and how they are integrated in the control-flow.

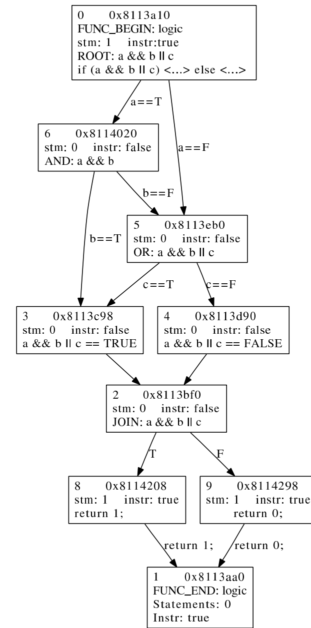
When a logic AND (&&) and OR (||) expression or a conditional expression operator (? :) is encountered during the CFG generation, an additional subgraph within the control-flow graph is created. Figure 4.3b shows the empty subgraph in step 1. These nodes can also be seen as nodes 0, 2, 3 and 4 in figure 4.3c. In the next step the expression is evaluated from left to right. The logical “and” is put in the structure of the CFG as it can be seen in figure 4.3b step 2. The next logical sub-expression, which is the “or”, is placed into the triangle T-1-F as seen in step 3. By evaluating the logical expression in the right order, which is given by the syntax tree, and including each logical AND, OR and conditional expression in the lower triangle of the created sub-CFG the control-flow of logic expressions can easily be

```
1 int logic (int a, int b, int c)
2 {
3   if (a && b || c)
4     return 1;
5   else
6     return 0;
7 }
```

(a) Code Example



(b) Modelling the CFG of `a&&b||c`



(c) Annotated Control Flow Graph (CFG)

Figure 4.3: Shortcut Paths in Logical Expressions

modelled. The syntax tree considers braces and operator precedence implicitly by applying the C grammar rules.

The analysis tool examines only logical AND and OR expressions. Therefore the T node and the F node do not necessarily represent that the whole expression evaluates to true or false at this control-flow point. Therefore the merge node after the sub-CFG of the logical expression is needed. Figure 4.3c shows an annotated CFG of subfigure a. It can be seen that the nodes contain no statements (*stm:0*) which makes instrumentation impossible, which is annotated as (*instr:false*).

For random test data generation and for the model checker model the logical subexpressions have to be evaluated. During the creation of the sub-CFG, each node is assigned a string which contains a logical expression. Node 6 holds `"(a)"`, node 5 holds `"!(a&&b)||(!a)"`, node 3 holds `"!(a&&b)||(c)"` and node 4 holds `"!(a&&b||c)"`. Node 2 is assigned `"1"` since it is always executed. If the model checker should cover the sub-path (6,5,3,2) the statement `if(!((a)&&(!(a&&b)||(!a))&&(!(a&&b)||(c))&&(1))) return;` is created. This makes the assertion used for test data generation unreachable on a different

```
1  int logic (int a, int b, int c)
2  {
3    if (a && b || c) {
4      if ((a)) instr(6);
5      if (!(a&&b)||(!a)) instr(5);
6      if (!(a&&b)||(c)) instr(3);
7      if (!(a&&b||c)) instr(4);
8      if (1) instr(2);
9      instr(8);
10     return 1;
11   } else {
12     if ((a)) instr(6);
13     if (!(a&&b)||(!a)) instr(5);
14     if (!(a&&b)||(c)) instr(3);
15     if (!(a&&b||c)) instr(4);
16     if (1) instr(2);
17     instr(9);
18     return 0;
19   }
20 }
```

Figure 4.4: Instrumentation of Logic Expressions

execution path through the logic expression. The code for the test data generation is similar. In each side of the *if*-statement the instrumentations for the nodes in the expression are placed as shown in figure 4.4. It should be noted that the instrumentation points for node 0 and 1 are not shown in figure 4.4 since they are in the calling function directly before and after the call to the *logic* function. More information about how instrumentations for test data generation and for model checking work can be found in section 4.13.

4.7 Const Analysis

The const analysis is required for the categorization of different loop types in the loop analysis, which is the next analysis step. Figure 4.5 shows a pseudo language implementation of the basic algorithm used for the simplified data flow analysis. Each CFG node holds the const variables at the beginning and at the end of the CFG which are stored in the class member variables `const_in` and `const_out`. Additionally the member variable `const_written` holds a list of the variables which are written in the CFG node. This is necessary to remove variables from the const list which are in the `const_out` list of the parents but assigned different values in each parent node. For instance after the statement `if(x) a=4; else a=4;` the variable “a” would not be considered as const since it has been updated separately in the parent CFG nodes.

Figure 4.5 does not include const analysis for loops. However this is a simple fixed point iteration which is performed until there are no more changes of the set of const variables in the loop header. When a loop header is encountered the set of constant

variables in the loop header is memorized. If the control-flow takes the backedge after the loop body the new set of const variables is calculated. If the memorized variable set differs from the actual const variable set the new const variable set is memorized and the loop body is processed again until a fixed point is reached. Iteration variables are handled specially. If the initial assignment to the iteration variable is a constant value and the iteration expression contains a constant value then the variable is considered as const, as long as it is not modified in the loop body by a non-constant variable.

4.8 Loop Analysis

The loop analysis consists of two steps. The first step is the categorization of the loop type based on data dependencies. The second step, which is not required for all cases is the calculation of the loop bound using model checking.

4.8.1 Loop Categorization

The explanation in this section assumes a simple "for"-loop with an initial expression statement, a loop check condition and an iteration expression like

```
for(i=0; i<=10; ++i),
```

where `i=0` is the initial expression, `i<=10` the check condition and `++i` the iteration expression. The analysis works for other loop types too but the loop categorization can be easier explained based on a single loop type.

The loop header and body are examined separately for control-flow dependencies. First the loop categorization examines if the initial expression and the iteration expression depend only on constant expressions and if the iterator variable is modified inside the loop body. Further the loop condition is checked to determine if the loop bound is constant or variable. Last the control-flow inside the loop is evaluated by scanning the condition of each conditional statement in the loop body and inlined functions in the loop body. Loops containing non-inlined function calls are considered to have control-flow dependent paths.

Based on the const analysis of the loop header and body the loop can be categorized in one of the three following categories. The selection of the measurement technique for the loop is then based on this categorization.

The first loop type has only constants in the loop header and only constant control-flow decisions in the loop body, therefore the loop has only a single execution path and a constant number of iterations (*SP/CI*). Since the loop does not have any control-flow decisions it can be replaced by a basic block for further analysis. This is done by removing all CFG information inside the loop body so that only the CFG containing the loop header remains. Since the code generation is based on the syntax tree the code will still be generated correctly for test data generation and measurements.

```
analyze_const_variables (cfg *fstart)
    // analyze global constants, and parameter files
    set<variable*> global_const_vars
    add_const_vars_defined_in_parameter_file(&global_const_vars)
    add_variables_with_const_storage_class(&global_const_vars)
    set<cfg*> processed
    queue<cfg*> waiting
    // analyze function start, add children to queue
    analyze_const_in_cfg_node(fstart, global_const_vars)
    processed.add(fstart)
    if (fstart->left_child) then waiting.push_back(fstart->left_child)
    if (fstart->right_child) then waiting.push_back(fstart->right_child)

    while (! waiting.empty())
        cfg *act = waiting.pop_front()
        bool parents_finished = true
        set<variable*> const_vars = parent[0].const_out
        // process each parent
        for each parent in act->parents do
            // intersect const variables of parents
            const_vars.intersect(parent.const_out)
            // for >2 parents remove updated variables
            if (parents.count() >= 2) then const_vars.subtract(parent.const_written)
            // check if all parents have been processed
            if (! processed[parent]) then
                parents_finished = false
                act_pathcount += pathcount[parent]
        if parents_finished then
            // process current node, add childs
            analyze_const_in_cfg_node(act, const_vars)
            processed.add(act)
            if (act->left_child) then waiting.push_back(act->left_child)
            if (act->right_child) then waiting.push_back(act->right_child)
        else
            // process current node later
            waiting.push_back(act)

analyze_const_in_cfg_node (cfg* node, set<variable*> const_vars)
    node->const_in = const_vars
    // process each basic block
    foreach bb in node->basic_blocks do
        // find lhs and rhs variables (including sideeffects)
        set<variable*> lhs = get_lhs_variables(bb)
        set<variable*> rhs = get_rhs_variables(bb)
        // if all rhs values are const the result is const
        if is_subset(rhs, const_vars) then
            const_vars.add(lhs)
            node->const_written.add(lhs)
        else
            const_vars.subtract(lhs)
    // write the set of new constants to the CFG node
    node->const_out = const_vars
```

Figure 4.5: Const Analysis

The second supported loop type does not have any control-flow decisions based on non-const variables in the loop body but the loop check condition uses non-const variables. If the initial expression or the iteration expression depends on non-const values too, the loop does not match this category. Based on the analysis this loop type is supposed to have a single control-flow path but a variable number of iterations (*SP/VI*). Like the *SP/CI* loop this type of loop can be reduced to a single CFG node, but test data has to be generated in order to execute the loop with the maximum iteration counter.

The last category does not make any assumptions on the number of iterations or control-flow decisions. It is designed to handle generic loops with multiple control-flow paths and variable iteration counts (*MP/VI*). In order to perform measurements test data has to be generated to cover each path inside the loop for each iteration. The WCET of the loop is then calculated as $\sum \max(\mathrm{ET_i})$ over all iterations. This approach handles loops which execute paths based on the iteration counter like "`if (i%2) ...`" well but it overestimates loops which execute paths based on the input data. An example would be a linear search over an array which performs some actions when a given key is found. In this case the test data generation would search the test data to execute the special case in each iteration. Section 5.3.3 examines this behavior and shows that the overestimation can be quite high. When conducting the measurements with the generated test data, two instrumentation points are placed after the loop header. In figure 4.1b the loop header is located in the CFG node 10 and the instrumentations are placed before node 8 and node 9. When the loop is reached by the control-flow, either the instrumentation in node 8 or node 9 is triggered, if the loop is not executed at all or if the first iteration is performed. The next measurement starts at node 9 and stops at node 8 when the loop condition is false or at node 9 when another iteration is performed. For loops with a contain more control-flow paths than the path bound segmentation of the loop body is performed and additional instrumentation points are placed inside the loop body.

4.8.2 Loop Bound Analysis

In order to perform measurements for *SP/VI* and *MP/VI* loops the loop bound has to be known. To calculate loop bounds the *CBMC* model checker is used. Figure 4.6 shows an excerpt of the application example shown in figure 4.1. The modifications needed for loop analysis can easily be performed in the code generator. The necessary alterations are the insertion of the loop counter at the beginning of the program, shown in line 1 of figure 4.6, the increment of the loop counter as first statement inside the loop, shown in line 22, and the assertion directly after the loop as shown in line 33.

After the needed modifications have been made the *CBMC* model checker is used on the model to check is the maximum iteration count of the loop is less than `LOOP_MAX`. The constant `LOOP_MAX` can be defined on the command line using the `-D` switch. The output of *CBMC* is shown in figure 4.7 for two subsequent calls, the first with a path bound of 4 and the second with a path bound of 3. Since the

```
1   int loopcounter = 0;
2   int a, b, c, j;
13
14  int mymain()
15  {
16    int i;
17    int k = j-j+2;
20
21    for (i=1; i<=5; ++i) {
22      loopcounter++;
23      if (i == 1) {
24        i += k;
25      } else {
26        if (i % 2) {
27          a++;
28        } else {
29          b++;
30        }
31      }
32    }
33    assert(loopcounter < LOOP_MAX);
57
58  }
```

Figure 4.6: Model Checking for Loop Bound Calculation

assertion does not hold in the second case, the verification of the model fails and the model checker generates a counter example. The model checker can be instructed to generate xml output which is easier to parse but the plain text output is better for a demonstration of the method. Therefore only the text output is not shown in figure 4.7. It is easy to perform a binary search on the loop bound, starting with increasing values LOOP_MAX$=1 \ldots 2^n$ until the verification model holds. Then a binary search can be performed on the interval $[2^{n-1} \ldots 2^n]$ until the loop bound is found. The search uses a maximum of $2 * \mathrm{ld}(n)$ steps. It is also possible to use more than one assertion at the same time and the method has also been implemented counting up in $1 \ldots 2^{8n}$ steps and down using 2^8 search intervals instead of 2. However, since the model checking process is fast enough, it is sufficient to use a binary search.

4.9 Counting Paths Between Two Nodes

In order to perform segmentation, the number of paths between two CFG nodes must be known. This section introduces the path counting algorithm used in the WCET analysis tool.

The algorithm shown in a pseudo-language implementation in figure 4.8 works by "executing" the control-flow nodes and calculating the sum of paths to all predecessor nodes which is the number of paths to the actual node. When control-flow joins are encountered the algorithm processes all other nodes until each parent node has been

```
$ cbmc --function mymain -D LOOP_MAX=4 example.c
file example.c: Parsing
Converting
Type-checking example
Generating GOTO Program
String Abstraction
Pointer Analysis
Adding Pointer Checks
Starting Bounded Model Checking
Unwinding loop 0 iteration 1 file example.c line 21 function mymain
Unwinding loop 0 iteration 2 file example.c line 21 function mymain
Unwinding loop 0 iteration 3 file example.c line 21 function mymain
size of program expression: 46 assignments
Generated 1 claims, 0 remaining
VERIFICATION SUCCESSFUL

$ cbmc --function mymain -D LOOP_MAX=3 example.c
file example.c: Parsing
...
Generated 1 claims, 1 remaining
Passing problem to MiniSAT
Running MiniSAT
51 variables, 26 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
Building error trace

Counterexample:
State 1 file example.c line 1 thread 0
----------------------------------------------------
  loopcounter=0 (00000000000000000000000000000000)
...
State 38 file example.c line 21 function mymain thread 0
----------------------------------------------------
  example::mymain::1::i=6 (00000000000000000000000000000110)

Violated property:
  file example.c line 33 function mymain
  assertion
  loopcounter < 3

VERIFICATION FAILED
```

Figure 4.7: CBMC Output for Loop Bound Checking

processed. For real applications in C++ it is convenient not to access the cfg nodes directly but to use small proxy classes which provide access functions to the left and right child and to the vector of parent nodes. This makes the algorithm useable in the forward and in the backward direction.

```
bigint count_paths (cfg* from, cfg* to)
   map<cfg*,bigint> pathcount
   queue<cfg*> waiting
   pathcount[from] = 1

   if (from->left_child) then
      waiting.push_back(from->left_child)
   if (from->right_child) then
      waiting.push_back(from->right_child)

   while (! waiting.empty())
      cfg *act = waiting.pop_front()
      bool parents_finished = true
      bigint act_pathcount = 0
      foreach parent in act->parents do
         if (! pathcount[parent]) then
            parents_finished = false
            act_pathcount += pathcount[parent]
      if parents_finished then
         pathcount[act] = act_pathcount
         if (act == to) then
            if (! waiting.empty()) then
               return -1
            else
               return act_pathcount
         if (act->left_child) then
            waiting.push_back(act->left_child)
         if (act->right_child) then
            waiting.push_back(act->right_child)
      else
         waiting.push_back(act)
```

Figure 4.8: Path Counting Algorithm

As a side note it should be mentioned that the same algorithm can also be used for end-to-end path calculations. Loops are counted to be executed with zero or one loop iteration. However this requires special handling since the algorithm waits at the loop header for the control-flow to merge. Since one of the parents is the backedge a deadlock would occur. Therefore loops are marked with a special flag and are executed when all parents which are not backedges have been evaluated. In the first iteration the loop body is processed using the sum of all parent path counts. When the control-flow reaches the loop header again, the number of paths of the loop is calculated from the sum of all parent path counts, including the backedges, and processing continues with the CFG node following the loop exit. The loop handling is not shown in figure 4.8 to keep the basic algorithm small.

4.10 Dominator and Postdominator Tree

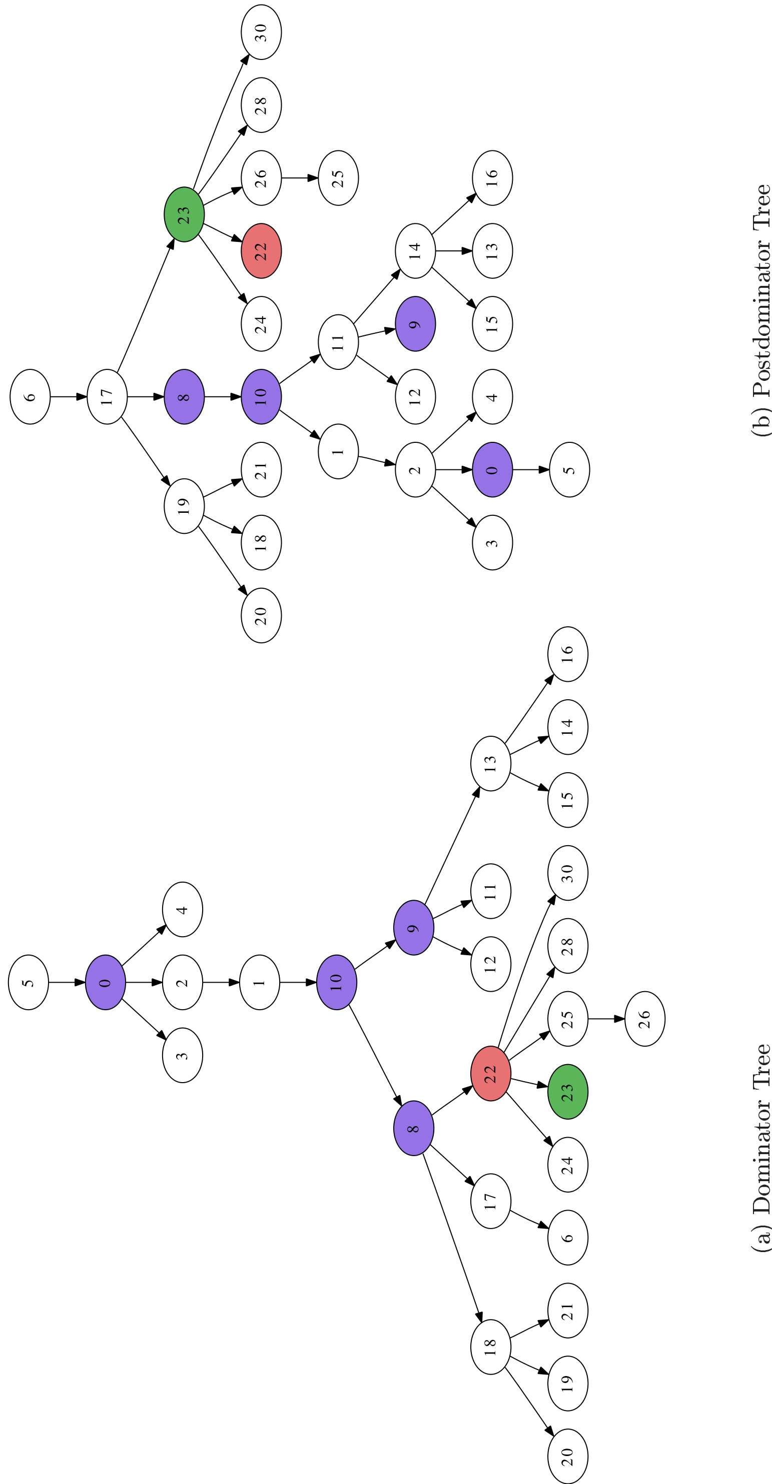
The dominator tree and postdominator tree represent control-flow hierarchies between basic blocks as defined in section 2.2.6. The root of the dominator tree starts from the entry node of the function. Each node in the tree is a dominator of all of its child nodes and their branches. Similar to the dominator tree each node in the postdominator tree postdominates its own children and their subtrees. The trees can be built using the control-flow but it is much easier to build them during the creation of the CFG and use structural knowledge about control-flow branches and merges that are available at this time.

Figure 4.9 shows the dominator and postdominator tree for figure 4.1b. The dominator and postdominator relations are very important for the segmentation process which is described in the next section. The definition of a program segment in section 2.2.7 states that there is only one entry node and only one exit node. Given a program segment $PS_i = (N_i, E_i, S_i, T_i)$ with $S_i = \{s_i\}$ and $T_i = \{t_i\}$ then $s_i = \text{dom}(n)$ must hold for all $n \neq s_i \wedge n \in N_i$ as well as $t_i = \text{pdom}(n)$ must hold for all $n \neq t_i \wedge n \in N_i$. This means the entry point is a dominator and the exit point is a postdominator for all program segments. Node 22 and node 23 which are specially marked in figure 4.9 show this relation for a particular program segment. Since $\text{BB}_{22} = \text{dom}(\text{BB}_{23}) \wedge \text{BB}_{23} = \text{pdom}(\text{BB}_{22})$ holds, there exists a PS with BB 22 as entry and BB 23 as exit point. Given a CFG $G = (N, E, s, t)$ a PS can be defined as $PS_3 = (N_3, E_3, \{s_3\}, \{t_3\})$ where $PS_3 \subseteq G$ and $\forall n \in N : [\text{BB}_{22} = \text{dom}(n) \wedge \text{BB}_{23} = \text{pdom}(n) \wedge n \in N_3] \vee [n \notin N_3]$. Generally all basic blocks in the program which are dominated by the start node of a program segment and postdominated by the exit node belong to the given PS. The other marked nodes, 0, 8, 9 and 10 are segment borders which have been set according to syntactic rules during the CFG generation and have to be considered during the segmentation.

4.11 Segmentation

The segmentation is based on the dominator and postdominator tree and the path counting algorithm introduced in the previous section. Segmentation uses the function *find_next_match* to find CFG nodes which can be used to create a PS so that $p = \text{dom}(\text{find_next_match}(p)) \vee \text{find_next_match}(p) = \text{pdom}(p)$. Additionally the function *check_for_marked* is used to check if a node in a possible PS requests segmentation. The path counting function *count_paths(x,y)* which has been defined in figure 4.8 is used to calculate the size of program segments.

The segmentation algorithm starts by finding a possible match (as defined above) to create a PS. If no match has been found, the function end or a control-flow join has been reached, assuming structured programming. In this case a PS is created with a single CFG node. If a match has been found the algorithm tries to extend the PS until the path bound is reached (*count_paths(x,y)* >pathbound) or a CFG node



(a) Dominator Tree

(b) Postdominator Tree

Figure 4.9: DOM and PDOM tree for example.c

```
perform_segmentation (cfg* start, cfg *stop, function* f)
    cfg* match_cfg = find_next_match(start)
    // last segment of branch or function
    if (!match) create_segment(start, [stop], f, 1)
    else
        bool extend
        bigint match_paths = count_paths(start, match)
        bool match_marked = check_for_marked(start, match)
        do // try to grow the segment as large as possible
            extend = false
            if (match_paths <= pathbound && !match_marked) then
                cfg* next_cfg = find_next_match(match)
                bigint next_paths = count_paths(start, next)
                bool next_marked = check_for_marked(start, match)
                if (next_paths <= pathbound && !next_marked) then
                    match_cfg = next_cfg
                    match_paths = next_paths
                    match_marked = next_marked
                    extend = true
        while extend == true
        if (match_paths > pathbound || match_marked)
            // split the segment
            create_segment(start, [start->left_child, start->right_child], f, 1)
            perform_segmentation(start->left_child, match_cfg, s)
            if (start->right_child) perform_segmentation(start->right_child, match_cfg, f)
            perform_segmentation(match_cfg, stop, f)
        else // add the segment
            create_segment(start, [match_cfg], f, match_paths)
            perform_segmentation(match_cfg, stop, f)

// find a cfg t that s=dom(t) and t=pdom(s) and ensure t can be instrumented
find_next_match (cfg* s)
    queue<dom*> q;
    dom *s_d = bb2dom[s->id]
    pdom *s_p = bb2pdom[s->id]
    q.push_back(s_d->getChilds())
    while not q.empty()
        dom *t_d = q.pop_front()
        pdom *t_p = bb2pdom[dom->cfg_id]
        cfg *t = int2cfg[dom->cfg_id]
        // check if s=dom(t) and t=pdom(s) and if t can be instrumented
        if (is_parent(s_d,t_d) && is_parent(t_p,s_p) && t->instr) return t;
        else q.push_back(t_d->getChilds())

// check if a node in the possible PS requests segmentation
bool check_for_marked (start, match)
    if (start->request_segmentation)
        return true;
    return check_for_marked (start->left_child, match) ||
           (start->right_child && check_for_marked (start->right_child, match);
```

Figure 4.10: Segmentation Algorithm

is marked as a segmentation border. If one of these events occurs the size of the PS is not further increased.

In the case the path bound is reached or a CFG node is marked as program segment border for the first call to *find_next_match* then a PS consisting of a single node is created and the segmentation continues with the left and the right child nodes. The segmentation continues with the next node after the segment by calling *perform_segmentation* with cfg where the control-flows from left and right child nodes are merged.

The results of segmentation of figure 4.1a can be seen in subfigure b. The graphical representation of the program segments has been manually edited since the *dot*-tool from the *GraphViz* package does not draw them very nicely. In fact it is impossible to create a circular line from node 9 to node 9 which covers the whole PS4. This influences only the graphic representation of PS but has no effect on the analysis. Experiments concerning the segmentation for different path bounds can be found in [Wen06].

4.12 Decision Tree (dtree)

The *decision tree (dtree)* represents all possible control-flow decisions within a program segment. The root of the dtree represents the start of a PS. Each decision in the application creates a branch in the dtree. Each leaf of the dtree represents a unique execution path through a program segment. Figure 4.11 shows the dtree for the example application. The root of the tree representation is the analyzed function, followed by the program segments, which represent the first generation of child nodes. It can be seen in PS4 that each iteration of a MP/VI loop has its own dtree, or set of dtrees if the loop contains more than one basic block. The dtree is especially useful to hold test data. When each basic block in instrumented with its own ID then it is easy to monitor the execution from the root to a leaf. When using random test data and following the execution to the leaf, it can be seen whether test data exists already for a specific leaf or not. In the latter case a reference to the data set is stored in the dtree; further an indicator which marks the data as random generated. Additionally a counter which is initialized to 0 before each execution of the application and increased with the start of each encountered PS is stored in the leaf of the dtree, denoted as n=... in figure 4.11. This is important when performing measurements and the instrumentation points carry only a time stamp and no ID as described in section 4.15. In addition all leafs of the dtree are connected as a linked list, which makes it easy to check which leafs hold already test data and which are empty.

After test data generation, which is described in the following section, each leaf node of the dtree should be reached by random data *RND*, model checker generated data *MC* or be infeasible *INF* and carry a reference to a data set *ds* as well as a PS counter *n*. Figure 4.11 shows the dtree of the example application when 100 random data sets are used. For all experiments conducted in chapter 5 10,000 random

Figure 4.11: DTree for the Example Application

data sets have been used. In order to observe how the model checked test data are integrated with the random test data this number has been reduced. It can also be seen that many paths in the loop are infeasible since the loop is basically a single path loop and the analysis framework has been tricked to treat it as MP/VI loop through the use of an unknown variable as shown in figure 4.1a line 16.

A design flaw in the decision tree implementation of the new prototype implementation is that only the children of the root node may contain iteration counters for loops like PS4 in figure 4.11 and hold child nodes for individual iterations. This makes it impossible to represent nested loops in the dtree. Basically it would not be challenging to change the underlying data structures to allow nested loops but since the dtree is a central data structure a huge number of changes in all modules would be required, with an estimated implementation effort of four weeks (including testing and debugging).

4.13 Test Data Generation

The dtree is used to find out which test data set leads to the execution of a given CFG path. By using instrumentations on all basic blocks, the dtree is also useful to follow the execution path of the application. And the dtree can be used to create the model checker model. The following sections describe how test data is generated and how the dtree is used in this process. All individual test data generation methods can be disabled by command line options: Test data reuse and model checking directly by command line switches, random test data generation by setting the number of random generated test data sets to zero.

4.13.1 Reused Test Data

The first step when generatig test data is to reuse existing test data. When measurements are performed all test data are stored in an xml file. On successive measurements the saved test data are loaded from the xml file and reused. It would possibly be a good idea to include a representation of infeasible paths in the test data cache too, but then there has to be ensured that the application is exactly the same, and that the segmentation and other command line options remain exactly the same for each analyzer run. A similar effect can be achieved by disabling model checking from the command line and using cached data, since test data for the execution of all feasible paths is stored in the cache. However, this should be handled with care.

The test data cache cannot be used to measure a set of user generated test data, since only one data set for each execution path in a PS is used. The rest is discarded. Therefore it would be difficult to assign the measurement results to a unique data set.

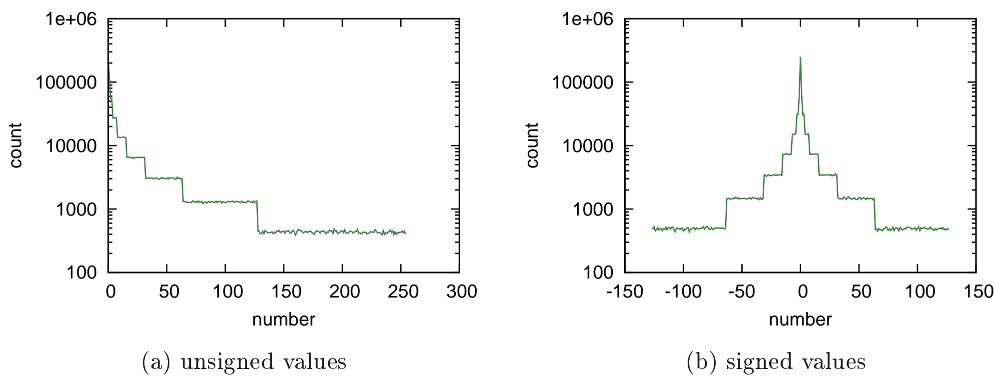
4.13.2 Random Test Data

Random test data generation uses a random number generator to generate the test data. The random seed can be specified on the command line. Since the first version of the *MoDECS* prototype which was barely usable was executeable for the first time on Christmas eve the random seed is traditionally set to 2412. The random number generator uses two random numbers to generate a single unsigned random number and three to generate a signed random number (see below).

The first number k which is generated defines the width of bits which should be used for the random number. Only the lower k bits of the second random number are used. The third random number is used to determine the sign for signed data types. The two random numbers can be combined to a single n bit wide random number using

```
random() & ((1 << (random() % <n+1>)) - 1)
```

where $n \in \{8, 16, 32\}$. The goal is to modify the even distribution of random numbers to accumulate around 0 and fall off with increasing positive and negative values as shown in figure 4.12 for 1,000,000 generated unsigned and signed 8 bit random numbers. The idea behind this distribution is that applications often use flags or enums which hold normally small values or zero.



(a) unsigned values (b) signed values

Figure 4.12: Random Value Distribution for Test Data Generation

To identify the execution paths which are taken, an instrumented version of the target application code is generated. Each basic block is instrumented with a call-back function that is called to log the ID of the basic block when executed. The application is compiled and dynamically loaded and linked to the WCET analysis framework. Therefore there may not be a function named "*main*" in the analyzed source code since this prevents the module from loading correctly. When the instrumented application is generated a small stub-function is included in the source file which contains instrumentation points immediately before and after the analyzed function. The stub function is also responsible for assigning the generated test data to the input variables and static variables before executing the function under test. The mechanism which is used to write the variables is explained in section 4.15 since it is the same algorithm that is used in the created target code. Each time the func-

tion is executed the execution path is traced and compared to the current position in the dtree. If a leaf node is reached, which is not already covered by a different test data set, the current test data set and PS counter n are assigned to the leaf node.

4.13.3 Model Checking

Not all paths can easily be covered by random number generators. Nested conditions are hard to cover. Increasing the number of random generated test data sets can improve the coverage a little, but this does not solve the problem. There are infeasible paths in the dtree. Only model checking or an other formal method which is guaranteed to be complete can prove that these paths are really infeasible and have not only been missed during the previous test data generation steps. Model checking for test data generation works by enforcing the execution of a given path within a PS. The assertion used to verify the application is that the given path cannot be executed. If the path can be executed then the assertion fails and the counter example contains the test data to reach a given path. The execution of a given path can easily be forced by introducing a new variable, *mc_path* in line 2 of figure 4.13, which is increased at the beginning of each node which lies on the execution path. At the end the variable is asserted not to be equal to the number of nodes on the path. If the number is equal to the nodes the assertion fails and the test data can be acquired from the counter example.

The model checker based test data generation can be extended to work on loops too. To do this a loop counter variable is introduced and the loop counter is used to act as guard expression to the modification of the variable used to verify the execution path as shown in figure 4.13 in line 24, 35, 27, etc. To create test data covering control-flow decisions in statements the annotated conditions along the selected control-flow path are used as guards to increase the *mc_path* variable. If both, loops and control-flow in statements are used at the same time, the modification of *mc_path* is guarded by the loop counter and the annotated expressions.

When the model checker finds a counter example, the test data can be easily extracted from it. The model checker output looks similar to figure 4.7; therefore it is not shown here. However, the framework uses the xml output since it can easier be parsed. The test data generation uses the list of dtree leaf nodes and creates a model for each leaf without a data set assigned to it. After the model checking has completed each dtree node without a test data set is guaranteed to be infeasible.

4.14 Managing Test Data

The test data are held in a contignuous memory area. Therefore the `-s` command line parameter cannot be arbitrarily high. However, the gained coverage of an increased number of test data sets is only marginal for high values of `s`, which is more than 100.000–1.000.000 test data sets. During the analysis process a data structure is created which allows access to all variables based on the data set ID and the variable

```
1  int loopcounter = 0;
2  int mc_path = 0;
3  int a, b, c, j;
14
15 int mymain()
16 {
17   int i;
18   int k = j-j+2;
21
22   for (i=1; i<=5; ++i) {
23     loopcounter++;
24     if (loopcounter==1) mc_path++; /* BB10 */
25     if (loopcounter==1) mc_path++; /* BB9 */
26     if (i == 1) {
27       if (loopcounter==1) { mc_path=-1; } /* BB12 */
28       i += k;
29     } else {
30       if (loopcounter==1) mc_path++; /* BB13 */
31       if (i % 2) {
32         if (loopcounter==1) mc_path++; /* BB15 */
33         a++;
34       } else {
35         if (loopcounter==1) mc_path=-1; /* BB16 */
36         b++;
37       }
38       if (loopcounter==1) mc_path++; /* BB14 */
62       default:
63         a=b;
64     }
```

Figure 4.13: Model Checking to find a Path inside the Loop

ID or to a whole test data set in binary format. The test data class, which is named *tdata*, can read and write xml files containing the test data like the example shown in figure 4.14. Only the test data which are used in the current analysis remain in the data base except for test data which have been loaded from the cache but are not used during the current analysis. The test data class also contains a code generator, which can create variable declarations for static variables as well as assignments from a binary representation of the test data to variables. This is used to set the function parameters and static variables when executing instrumented code.

4.15 Measurements

In contrast to the *MoDECS* WCET analysis prototype the new implementation can handle multiple instrumentation points in a single run, which is required for loops. Therefore there is no need to recompile a different version of the target code for each program segment. This increases the speed of the measurement process considerably.

The term measurement in this chapter does not only apply to measurements on the target hardware but also to the execution of an instrumented version of the

```
1  <function name="mymain" cSource="../testcases/example.c">
2    <variable id="0" name="a" size="32" type_name_host="signed int" offset="0" global="true" />
3    <variable id="1" name="b" size="32" type_name_host="signed int" offset="0" global="true" />
4    <variable id="2" name="c" size="32" type_name_host="signed int" offset="0" global="true" />
5    <variable id="3" name="j" size="32" type_name_host="signed int" offset="0" global="true" />
6
7    <recordset uniqueDatasets="14">
8      <record id="0" ps="1" n="1" fname="example.target.c">
9        <valList>
10         <val>2</val>    <!-- name: a -->
11         <val>3</val>    <!-- name: b -->
12         <val>0</val>    <!-- name: c -->
13         <val>0</val>    <!-- name: j -->
14       </valList>
15     </record>
16     <record id="0" ps="2" n="2" fname="example.target.c">
17       ...
```

Figure 4.14: Example xml Data File

application on the host in order to generate test data. This section explains different aspects of code generation and how the test data can be applied on the measured application. How the actual code generation for the tests works is not explained here, since it is simply an in-order traversal of the syntax tree while printing the encountered literals or lexer symbols.

4.15.1 Generating Target and Module Code

In the previous section the use of a loadable module for random test data generation was described. Figure 4.15 shows a host loadable module with instrumentations created from the example application. The module is compiled using the *system* function and *gcc* with the command line parameters `-ggdb3 -shared`. When the compilation succeeds the module can be loaded using

```
dynlib = dlopen(obj_name.c_str(), RTLD_LAZY | RTLD_LOCAL);
```

which returns a handle to the dynamic library. The handle is required to determine the address of the *caller* function using the *dlsym* as shown below.

```
caller = (void (*)(int,void*,void (*)(int),void(*)(int))) dlsym(dynlib, "caller");
```

After all measurements have been performed the library can be unloaded using the *dlclose* function call.

```
dlclose(dynlib);
```

The functions *begin* and *end* are automatically executed since they use the attribute `constructor` respectively `destructor` in their declaration. After the module has been loaded using *dlopen* and the address of *caller* has been determined using *dlsym* the *caller* function can be executed. The function parameters are the number of data sets, the memory location of the input data and two call-back functions. One of them is called for each instrumentation and the other one is called each time after the execution of the analyzed function finishes with the number of the data set.

```
1   extern int printf(char *f, ...);
2   void (*instr) (int x);
3   void (*finish_data) (int ds);
4
5   int a, b, c, j;
6
7   int equal (int _a, int _b)
8   {
9     int rv;
10    if (a == b) {
11      instr(3);
12      rv == 1;
13    } else {
14      instr(4);
15      rv == 0;
16    }
17    instr(2);
18    return rv;
19  }
20
21  int my_main ()
22  {
23    int i;
24    int k = j-j+1;
25    instr(0);
26    c = equal(a, b);
27    instr(1);
28    instr(10);
29    for (i=1; i<=5; ++i) {
30      instr(9);
31      if (i == 1) {
32        instr(12);
33        a++;
34      } else {
35        instr(13);
36        if (i % 2) {
37          instr(15);
38          i += k;
39        } else {
40          instr(16);
41          b++;
42        }
43        instr(14);
44      }
45      instr(11);
46    }
47    instr(8);
48    if (a) {
49      instr(18);
50      if (b) {
51        instr(20);
52        a=0;
53      } else {
54        instr(21);
55        b=1;
56      }
57      instr(19);
58    } else {
59      instr(22);
60      switch (b) {
61        case 1:
62          instr(24);
63          a=1;
64          break;
65        case 2:
66          instr(25);
67          b=1;
68        case 3:
69          instr(26);
70          c=a;
71          break;
72        case 4:
73          instr(28);
74          break;
75        default:
76          instr(30);
77          a=b;
78      }
79      instr(23);
80    }
81    instr(17);
82  }
83
84  int caller (int datasets, void *data,
85      void (*_instr) (int),
86      void (*_finish_data)(int))
87  {
88    int ds;
89    char **signed_char_data =
90      (char **) (&data);
91    unsigned char **unsigned_char_data =
92      (unsigned char **) (&data);
93    int **signed_int_data =
94      (int **) (&data);
95    unsigned int **unsigned_int_data =
96      (unsigned int **) (&data);
97
98    instr = _instr;
99    finish_data = _finish_data;
100   finish_data(-1);
101   for (ds=0; ds<datasets; ++ds) {
102     a = *((*signed_int_data)++);
103     b = *((*signed_int_data)++);
104     c = *((*signed_int_data)++);
105     j = *((*signed_int_data)++);
106     instr(5);
107     mymain();
108     instr(6);
109     finish_data(ds);
110   }
111 }
112
113 attribute ((constructor))
114 void begin ()
115 {
116   printf("+++ module start\n");
117 }
118
119 attribute ((destructor))
120 void end ()
121 {
122   printf("#+++ module end\n");
123 }
```

Figure 4.15: Generated Loadable Module

Each test data set is extracted from a large memory area containing a binary representation of the test data. The code which reads the data and assigns it to the variables can be seen in lines 102 to 105 of figure 4.15. Since this method is very light-weight it can also be used on the target platform. The difference is that only enough memory for a single data set is created and that each data set is transferred individually from the host to the target. However, the mechanism to parse the binary data is the same. It uses a pointer to the test data. To advance the pointer by the right amount special pointers to *data* are used:

```
char  **signed_char_data   = (char **) (&data);
short **signed_short_data  = (short**) (&data);
int   **signed_int_data    = (int **)  (&data);
```

Using these pointers the values can easily be extracted from the binary data area since the *tdata* class holds information about the represented variables such as their name and datatype.

```
// data is advanced by 1 byte on next read operation
my_char   = *((*signed_char_data)++);
// data is advanced by 2 bytes on next read operation
my_short  = *((*signed_short_data)++);;
// data is advanced by 4 bytes on next read operation
my_int    = *((*signed_int_data)++);;
```

The *caller* function takes two function pointers. The first function is called for each occurrence of *instr(x);* in the module code. The second function, *finish_ data(n)*, is called after data set n.

It can also be seen that the first instrumentation is inserted before the call to the analyzed function and the last instrumentation is inserted after the function returns. This is not important for test data generation but it is important for execution-time measurements. By placing the instrumentation points in the calling function the overhead of the function call is included in the measured execution-time.

4.16 Analysis Tool Usage and Output

This section describes the usage of the WCET analysis tool. The tool has no graphical user interface and is designed for command line use only. To supply additional information for the analysis a parameter file can be used. If contradicting options are given in the parameter file and on the command line, the command line overrides the parameter file. This makes it possible to use a default setting but to test certain options without the need to edit the parameter file.

4.16.1 Parameter File

The parameter file has to have the same name as the first source file with the extension `.par` instead of `.c` and has to be located in the same directory. The

parameter file contains information on constant variables and on the handling of functions which can used during the analysis. It is possible to use regular expressions on the variable names. This is useful when the const keyword is used incorrectly and constants which are in an application ROM are declared as `volatile unsigned char x;`. This is the case in one of the industrial examples in chapter 5. Further it can be decided for functions if they should be inlined or black-boxed. When functions are to be black-boxed their WCET in cycles has to be specified, since the WCET analysis tool does not support calling itself recursively to estimate the WCET of black-boxed functions. A parameter file which is unrelated to the application example is shown in figure 4.16.

```
1 # treat all variables named r_... as const values
2 const r_.*
3 # treat all variables named C... as const values
4 const C.*
5
6 # use black-boxing on function my_func1 with WCET=15673 cycles
7 function my_func1 bb 15673
8 # perform inlining on function my_func2 (default)
9 function my_func2 inl
```

Figure 4.16: Parameter File

4.16.2 Command Line Arguments

Since the analysis tool has no user interface, all inputs have to be provided using the command line interface. Parameters on the command line have higher priority than commands in the parameter file. All parameters use reasonable default values so that the only argument which is really required is the name of a source file to be analyzed. Figure 4.17 shows the program's help output. Since the explanation of the parameters is given in the output they are not discussed in detail here.

Like in the parameter file the inlining or black-boxing of functions can be enabled on the command line. However there is no way to specify const variables on the command line. This can only be done in the parameter file. The command line can be used to disable some of the analysis features and observe how the analysis behaves without the use of these features. For instance it can be observed how often the WCET path is missed when disabling model checking and using only random data. On subsequent analysis runs on the application it is possible but not recommended to turn of model checking but only if the application has not changed.

It is important that all measurements can be reproduced. This does not only require the same target hardware but also the same version of the WCET analysis tool. Using `wcet -version` shows the CVS version of all source modules, tests and tools which were used when the analysis tool was built as shown in figure 4.18. The file `version.h` is modified each time the analysis tool is built. It contains the build

```
$ ./wcet --help
wcet vers: v3.0 alpha (build 477)
[-h|-help|--help]             .. print this help message and exit
[-r random-seed]              .. use given random seed (2412)
[-s populationsize]           .. use given number of random data sets (10.000)
[-p path_bound]               .. use given path bound for segmentation (100)
[-dp|--show-progress]         .. print the progress during processing (false)
[-da|--show-params]           .. print parameters in effect (false)
[-fi|--do-inlining] <f>       .. use function inlining for function f
[-nf|--no-inlining-func <f=cycle>] .. do not inline function f and use the
                                 given time in cycles for black-boxing
                                 (i.e. -nf reset=168:init=98:set_output=68)
[--no_expression_paths]       .. disable expression paths (false)
[-m|--disable-mc]             .. disable model checking (false)
[-c|--no-data-cache]          .. do not use stored test data
[--strict]                    .. exit on unstructured programming constructs
[-t|--target]                 .. set the target platform (arm_olimex)
                                 supported: arm_olimex
[-v|--version]                .. print version information and exit
[-f|--function <sf>]          .. the function to be analyzed (last function)
[--prefix <path>]             .. use path for intermediate and result files
file[.c] ...                  .. file(s) to be analyzed
```

Figure 4.17: WCET analysis Tool Help Text

number as well as the CVS version of the source file. Unless some files are out of syncronization with the CVS repository is it always possible to check out exactly the same state and reproduce a given test result.

4.16.3 Tool Output

The output of the tool when analyzing *example.c* can be seen in Figure 4.19. The option `-s 100` was selected to generate only 100 random test data sets so that some paths have to be searched using model checking. The options `-da` and `-dp` were selected to increase the verbosity of the tool. The first two output lines show the build number and the command line parameters. The third line informs the user that no parameter file has been found, which is not a problem. The following lines show the actions the tool is currently performing followed by details about the segmentation process. The terms "*fstart*" and "*fexit*" denote the segmentation borders are the function start and exit, "*PS marker*" means that the CFG generation caused segmentation at this point and "*PBound*", which is not shown in this example, means that the segmentation was caused by the path bound. The following lines show the continuing progress of the analysis up to the lines "*+++ module start*" and "*+++ module end*" which are printed from inside the loaded module. The next lines from line 27 to line 32 show the reached path coverage using random test data. In this case 11 of 30 paths have been covered, leaving 19 paths for the model checker. Line

```
$ ./wcet -v
********************************************************************************
WCET Analyzer Toolv3.0 alpha (build 477)
compiled at Mar 27 2009 15:48:00 using GCC 4.3.3
********************************************************************************
Build Time: 2009-03-27 16:53:25
.: ANSI-C.l                                                                [1.5]
.: ANSI-C.y                                                                [1.11]
.: Makefile                                                                [1.18]
.: bigint.cc                                                               [1.7]
.: bigint.hh                                                               [1.5]
...
.: variable.cc                                                             [1.8]
.: variable.hh                                                             [1.8]
M: version.hh                                                              [1.2]
.: tests/01_V3/00_tests.cfg                                                [1.1]
.: tests/01_V3/01_variables_and_data_types/01_global_variable_declaration.c   [1.1]
.: tests/01_V3/01_variables_and_data_types/02_local_variable_declaration.c    [1.1]
...
.: tools/filter_cvs_status.pl                                              [1.3]
.: tools/gdb_commands                                                      [1.1]
.: tools/increment_buildno.pl                                              [1.1]
```

Figure 4.18: WCET analysis Tool Version Information

number 34 with the dots and the pluses grouped in units of ten characters represents the progress of the model checker. Each dot represents a safe model, this means an infeasible path. A plus marks model checker runs for which the model is unsafe and test data has been found. The next lines from line 36 to line 41 show the path coverage after model checking. There are still 16 infeasible paths left. The following lines show the progress of the measurements. The stripped object, which has a size of 1444 bytes as shown in line 44, is downloaded and measurements are performed. Each dot represents a single measurement. After this the execution-times for the individual paths and the repetitions of the loop are displayed from lines 48 to 54. At last the execution-time is printed in line 56 which is 223 clock cycles for the example application.

4.17 Summary

In this chapter the individual building blocks of the WCET analysis framework have been discussed. The used algorithms have been explained using a C-like pseudo code language and the used data structures have been visualized using graphs for a given example. Finally the usage of the WCET analysis tool has been explained and the output for the analysis of the example has been illustrated.

```
1  $ ./wcet -s 100 -da -dp example.c
2  wcet vers: v3.0 alpha (build 477)
3  args: ../wcet --show-params --show-progress -s 100 ../tests/func.c
4  parameter file '../tests/example.par' not found
5  parsing input file ... done
6  building CFG ...
7    inlining equal
8  done
9  analyzing functions ...
10   main 56 paths, 2 PS markers
11 done
12 performing segmentation
13   PS1 -> 1 paths, border: fstart, PS marker
14   PS2 -> 2 paths, border: PS marker, PS marker
15   PS3 -> 2 paths, border: PS marker, PS marker
16   PS4 -> 6 paths, border: PS marker, PS marker
17   PS5 -> 7 paths, border: PS marker, fexit
18 done
19 building dtree ... PS1 PS2 PS3 PS4 PS5... done
20 analyzing constant flow ... done
21 calculating loop bounds 1,2,4,3 done
22 building module .. done
23 executing
24 +++ module start
25 using 100 random data sets
26 +++ module end
27 paths covered 11/30   ( 33%)
28   PS1   1/1   (100%)
29   PS2   2/2   (100%)
30   PS3   1/2   ( 50%)
31   PS4   3/18   ( 17%) @3x
32   PS5   4/7   ( 57%)
33 using MC on 19 paths
34   .......... ......+++
35 done
36 paths covered 14/30   ( 47%)
37   PS1   1/1   (100%) I:0
38   PS2   2/2   (100%) I:0
39   PS3   1/2   ( 50%) I:1
40   PS4   3/18   ( 17%) I:15 @3x
41   PS4   7/7   (100%) I:0
42 performing measurements
43 reseting target
44 downloading ... (1444bytes)
45 performing 14 measurements
46   .......... ....
47 done
48   PS1 .     max(ET)=32
49   PS2 ..     max(ET)=19
50   PS3 .     max(ET)=13
51   PS4@1 .     max(ET)=28
52   PS4@2 .     max(ET)=39
53   PS4@3 .     max(ET)=40
54   PS5 .......     max(ET)=52
55 calculating WCET
56 WCET = 223
```

Figure 4.19: Tool Output

Chapter 5

Experiments

This chapter describes the tests that were performed with the WCET analysis prototype. The next section describes the test environment, including the selection of test cases and the test hardware. The following sections will give a short introduction about the effect that is expected to be observed in the following test runs, and a detailed tabular listing of the actual test results. The three main topics in the process are inlining vs. black-boxing, different aspects of loops, and control-flow paths in expressions. The chapter concludes with a short section discussing the test results and a chapter summary.

5.1 Test Setup

This section describes the code metrics and values such as the number of basic blocks or the number of end-to-end paths that are used to describe the size and complexity of the case studies and the results in this chapter. The next sections describe the selection of test cases and the hardware as well as the software setup of the test environment, including the application download to the target and the host-target communication.

5.1.1 Basic Block and Path Counts Explained

Figure 5.1 is used to show how the code properties used in this section are calculated. These explanations are important to understand the tables of measurement results within this chapter. It can be easily seen that this example has 17 LOC and with a little counting a program size of 204 bytes (including whitespaces and line-breaks) can be determined. In the *main* function there is one basic block and one end-to-end path. In the *nsum* function there are four basic blocks. Each of the lines 4, 5, 6, and 7 forms an individual BB. The counting of paths is done at a time where loop bounds and flow facts are unknown. To be able to give a meaningful number for end-to-end paths the loop is assumed to have zero or one repetitions, which means the

loop creates a path for "no repetitions" and one path for "one repetition". Therefore we get a total of two end-to-end paths for *nsum*.

```
1  /* calculate sum of 1 .. n */
2  int nsum (int n) {
3    int i, sum;
4    sum = 0;
5    for (i = 1; i<=n; ++i)
6      sum++
7    return sum;
8  }
9
10 /* the main function */
11 int main()
12 {
13   int n;
14   n=4;
15   nsum(4);
16   return 0;
17 }
```

(a) Code Metrics Example

Bytes	204
LOC	17
BB	5
Paths EE	3
PS	4
Paths meas	4
Paths MC	0
Paths inf	0

(b) Code Properties

Figure 5.1: Code Metrics Example

The basic blocks (BB) and end-to-end paths ($Paths\ EE$) are summarized to get the numbers for the whole case study. Therefore we get a total of 204 bytes, 17 LOC, 5 BB and 3 Paths EE in figure 5.1.

However, during analysis more knowledge of the function is gained. It can be seen during analysis that the loop in *nsum* is unbounded. Therefore *nsum* has to be inlined. Additionally the data flow analysis tells us $n \leftarrow$ const and for the loop header $i \leftarrow$ const. Since all other values in the loop header are const the loop has a constant number of iterations. As the loop body is also variable independent, in fact it consists only of a single BB, the whole loop can be reduced to a single statement. Thus, after analysis only a single path remains to be measured. However, the segmentation still is determined by the number of syntactically existing paths and the occurrence of function calls and loops. Therefore we have still 4 program segments $PS = 4$. Since the end-to-end path that has to be measured lead through 4 program segments we have 4 paths to measure ($Paths\ meas = 4$). A single path is always covered by "random" data generation ($Paths\ rnd = 3$) and there are no paths that have been found using model checking ($Paths\ MC = 0$) or infeasible paths ($Paths\ inf = 0$). Consequently the time for generating test data using model checking ($T\ MC$) and the time for finding loop bounds ($T\ LB$) are zero.

It is important to note the difference between "*Paths EE*", which is the sum of all syntactical possible end-to-end paths and "*Paths meas*" which denotes the number of paths which are local to a single program segment and equals the number of paths that have to be measured to get a full coverage and a complete timing model of the case study.

5.1.2 Selection of Case Studies

The test cases used in this section are generic test cases that point out an expected effect and make it observable, the Mälardalen benchmark suite [Gro06] as well as three small to medium sized industrial applications from a former research partner during the *MoDECS* project.

The purpose of the generic test cases is to verify the presence of an expected effect and to observe the impact of this effect in a very simple setting so that it can be isolated from other interfering effects. Due to the fact that the described effect may change other application properties, such as the modification of program segment bounds or the total number of end-to-end control-flow paths, this may result in adverse effects, which compensate the described effect or even will reverse it. Therefore it is convenient when the test case is small enough to understand it and trace it manually.

The Mälardalen benchmarks are a set of mostly generic benchmarks which are used for WCET analysis. They are widely used in the WCET community, even if the majority of them is hand-written and they often use structures which are completely different from software generated applications. However, two of the benchmarks, *nsichneu* and *statemate*, are software-generated and resemble the industrial benchmarks described below in their structure. What makes the Mälardalen benchmarks interesting for the reseach-community is that they are available free of charge and not encumbered with usage restrictions. This means that anybody can use them for her/his own projects and publications or have a look at them when they are used as tests cases or examples in a publication.

The industrial examples, which are applications that are actually used in current car series, were kindly provided by a former research partner whose core business is the automotive industry. These test cases are generated by *TargetLink*® which is an addon to *Matlab/Simulink*® and widely used for model-based design in the (automotive) industry. As these test cases are the only applications investigated that are actually used in commercial embedded real-time systems, they are used whenever possible.

Figure 5.2 shows the code generation options used to generate the industrial examples. It is important that *Assembler statements* are turned off, since otherwise optimized assembler code would be generated for arithmetic functions, which are not supported by the target architecture. These optimized functions cannot be analyzed with a source code oriented analysis framework. However, the assembly code is not inserted directly into the source code by *TargetLink*® but inserted by means of macro definitions which are expanded using the C-preprocessor. These macro definitions contain the assembly code but can be replaced with functionally identical definitions for the test data generation. When performing run-time measurements the original macros can be used again. The *Clean code option* produces better human readable code when enabled but since the execution-time analysis framework is to be tested under realistic conditions (when possible), this option is turned off. *Inlining threshold* controls the size in statements up to which a series of statements is inlined

```
1  *** CODE GENERATOR OPTIONS:
2  *** Compiler                            : COSMIC44
3  *** Target                              : HCS12
4  *** ANSI-C compatible code              : yes
5  *** Optimization level                  : 2
6  *** Constant style                      : decimal
7  *** Clean code option                   : disabled
8  *** Logging mode                        : Acc. to block-spec. data
9  *** Linker sections                     : disabled
10 *** Assembler statements                : disabled
11 *** Variable name length                : 31 chars
12 *** Separate lookup search function     : disabled
13 *** Use global bitfields                : disabled
14 *** Stateflow: use of bitfields         : enabled
15 *** State activity encoding limit       : 5
16 *** Omit zero inits in restart function : disabled
17 *** Share fcns between TL subsystems    : disabled
18 *** Generate 64bit functions            : enabled
19 *** Inlining Threshold                  : 20
20 *** Line break limit                    : 100
21 *** Target optimized boolean data type  : enabled
22 *** Keep saturation elements            : disabled
23 *** Extended variable sharing           : disabled
```

Figure 5.2: Code generation options for industrial case studies

(below and equal to threshold) or a function containing the common code is created (above threshold). Last it is also obvious that the code was originally created for a *HSC12* processor using the *COSMIC®* compiler. As the WCET analysis framework is strongly oriented on ANSI-C code the option *ANSI-C compatible code* should also be turned on.

Other test cases were considered, like the *specint* test suite, but because of their program structure these tests have been prooven unusable as WCET benchmarks [Eng99].

In addition to the conducted tests there exist many regression tests that have been used during the development process of the timing analysis framework. This tests have been used to verify specific components of the analysis framework like the parser, the data flow analysis and the loop categorization algorithm. The test cases are all mostly very specific to certain problems or bugs which have occurred during the development. A small fraction of these regression tests is used to imitate code constructs which are commonly used by code generators. However, since there are now five machine generated test cases, there is no need to use the regression tests for experiments and execution-time measurements.

Table 5.1 shows an overview of all evaluated test cases. The first column is the test name, which is identical to the source module name. The second column denotes the source of the test case. *I* is used for industrial and *M* for Mälardalen benchmark suite.

Test Name	Src	Test Description	Bytes	LOC	BB	Paths
adpcm	M	Adaptive pulse code modulation algorithm. Completely well-structured code.	25977	878	125	73
bs	M	Binary search for the array of 15 integer elements. Completely structured.	4248	114	8	6
bsort100	M	Bubblesort program. Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers	2779	114	12	6
cnt	M	Counts non-negative numbers in a matrix. Nested loops, well-structured code.	2880	267	14	4
compress	M	Data compression program. Adopted from SPEC95 for WCET-calculation. Only compression is done on a buffer (small one) containing totally random data.	13411	508	63	324
cover	M	Program for testing many paths. A loop containing many switch cases.	5026	240	397	196
crc	M	Cyclic redundancy check computation on 40 bytes of data. Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.	5168	128	29	259
duff	M	Using "Duff's device" from the Jargon file to copy 43 byte array. Unstructured loop with known bound, switch statement	2374	86	26	12
edn	M	Finite Impulse Response (FIR) filter calculations. A lot of vector multiplications and array handling.	10563	285	47	21
expint	M	Series expansion for computing an exponential integral function. Inner loop that only runs once, structural WCET estimate gives heavy overestimate.	4288	157	26	13
fac	M	Recursive factorial calculation.	398	26	8	4
fdct	M	Fast Discrete Cosine Transform. A lot of calculations based on integer array elements.	8863	239	10	5
fft1	M	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm. A lot of calculations based on floating point array elements.	6244	219	56	33
fibcall	M	Simple iterative Fibonacci calculation, used to calculate fib(30). Parameter-dependent function, single-nested loop	3499	72	7	7
fir	M	Finite impulse response filter (signal processing algorithms) over a 700 items long sample. Inner loop with varying number of iterations, loop-iteration dependent decisions.	11965	276	15	6
insertsort	M	Insertion sort on a reversed array of size 10. Input-data dependent nested loop with worst-case of $(n^2)/2$ iterations (triangular loop).	3892	92	8	3
janne_complex	M	Nested loop program. The inner loops number of iterations depends on the outer loops current iteration number.	1564	64	18	15
jfdctint	M	Discrete-cosine transformation on a 8x8 pixel block. Long calculation sequences (i.e., long basic blocks), single-nested loops.	16028	375	13	6
lcdnum	M	Read ten values, output half to LCD. Loop with iteration-dependent flow.	1678	64	22	19
lms	M	LMS adaptive signal enhancement. The input signal is a sine wave with added white noise. A lot of floating point calculations.	7720	261	56	23

Table 5.1 “Description of Test Cases” is continued on the next page →

Test Name	Src	Test Description	Bytes	LOC	BB	Paths
ludcmp	M	LU decomposition algorithm. A lot of calculations based on floating point arrays with the size of 50 elements.	5160	147	50	45
matmult	M	Matrix multiplication of two 20x20 matrices. Multiple calls to the same function, nested function calls, triple-nested loops.	3737	163	15	11
minver	M	Inversion of floating point matrix. Floating value calculations in 3x3 matrix. Nested loops (3 levels).	5805	201	81	132
ndes	M	Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.	7345	231	38	143
ns	M	Search in a multi-dimensional array. Return from the middle of a loop nest, deep loop nesting (4 levels).	10436	535	14	7
nsichneu	M	Simulate an extended Petri Net. Automatically generated code containing large amounts of if-statements (more than 250).	118351	4253	1259	1.7e120
prime	M	Check if a number is a prime number. Loop, simple pointer operations	863	46	16	19
qsort-exam	M	Non-recursive version of quick sort algorithm. The program sorts 20 floating point numbers in an array. Loop nesting of 3 levels.	4535	121	39	63
qurt	M	Root computation of quadratic equations. The real and imaginary parts of the solution are stored in arrays.	4898	166	18	10
recursion	M	A simple example of recursive code. Self-recursion and mutual recursion.	620	41	14	13
select	M	A function to select the nth largest number in a floating point array. A lot of floating value array calculations, loop nesting (3 levels).	4494	114	39	86
sqrt	M	Square root function implemented by Taylor series. Simple numerical calculation.	3567	77	15	7
st	M	Statistics program. This program computes for two arrays of numbers the sum, the mean, the variance, and standard deviation, and the correlation coefficient between the two arrays.	3857	177	18	13
statemate	M	Automatically generated code. Generated by the STAtechart Real-time-Code generator STARC.	52618	1276	497	4.2e7
AktuatorMotorregler	I	Engine control application submodule from a former project partner and used in actual car series. Automatically generated, complex control-flow, loops	55749	1238	320	1.6e18
AdcKonv	I	Input/Output conversion submodule used donated from a former project partner and in actual car series. Automatically generated, no loops	15265	365	31	144
AktuatorSysCtrl	I	Engine control application submodule donated from a former project partner and used in actual car series. Automatically generated, no loops	14587	306	54	97

Table 5.1: Description of Test Cases

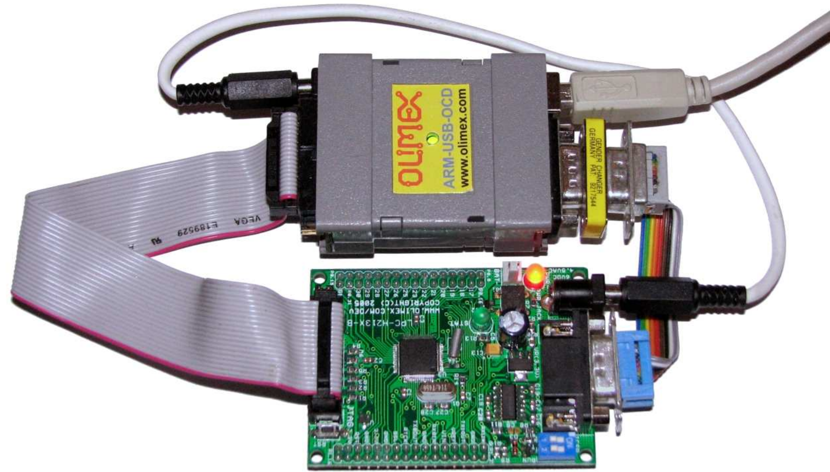


Figure 5.3: Olimex® LPC-H2138 Development Board and Programming Device

The third column gives a short functional and structural description about the test case, which is taken from [Gro06] for the Mälardalen benchmarks. The next columns give the size of the test case in bytes, lines of code (LOC) and basic blocks where only BBs containing expressions are counted (empty BBs created from logical expressions or control-flow joins are not counted in this column). The last column gives the number of end-to-end paths through the application, including paths generated by logical expression shortcuts.

5.1.3 Test Hardware and Development Software

All tests were performed on an Olimex® ARM evaluation board LPC-H3128 [OLI07] with a LPC-2138 Microcontroller. The LPC2138 contains a 16/32-bit ARM7TDMI-S CPU, 32k SRAM, 512k flash, two Universal Asynchronous Receive/-Transmit Units (UARTS) and two 32 bit timers and 47 general purpose I/O lines [PHI06]. The ARM core can either execute 32 bit instructions or variable length thumb instructions. Further features are a tiny size and low power consumption.

An Olimex® ARM-USB-OCD [OLI06] programming device was used to provide a JTAG interface to program the FLASH of the ARM. Additionally a USB-to-Serial interface and a power supply for the target hardware is provided by the programming device, which is connected to the host computer using a USB port.

Figure 5.3 shows the ARM evaluation board and the programming device. The blue DIP switch on the evaluation board can be configured to provide a special programming interface to the microcontroller which allows to program the device over the serial port. This is a very useful feature especially since the JTAG pins can be configured to work as normal IO pins and it is easily possible to disable the JTAG

interface when experimenting with the device. In this case the serial programming interface can be used to regain access to the device. The descriptions and schematics for both devices can be downloaded from the Olimex® web site.

The Olimex® prototype board and programming device were chosen because they provide a serial connection which makes the communication to the target system very easy. Also the hardware is well documented. The development environment works under Linux using the Open On-Chip Debugger (OpenOCD) [R+, Rat05] which can be used as a GNU debugger (gdb) server allowing remote debugging of the target using the JTAG interface, to program and verify the flash, to set breakpoints and to run the application.

The application was compiled with Version 4.3.3 of the GNU ARM Toolchain [Tea09] which consists of GNU binutils, GNU compiler set (GCC) including a C and a C++ compiler and debugger (Insight or GDB). Newlib is used for the C library. The command line optimization parameter used for the compiler was `-O0` to prevent control-flow modifications for all test cases except for *nsichneu* and *statemate* which were compiled with `-Os` to fit into the SRAM.

The flash provides a capacity of 512kB while the SRAM is only 32kB in size. There are two reasons for using the SRAM in favor over the flash. First, during the measurements each program segment gets instrumented separately, which causes at least as many downloading cycles as there are program segments in the application. Since the lifespan of a flash memory is limited to a few thousand erase cycles the microcontroller might soon die. Second, the LPC2138 provides a caching mechanism for the flash. This cache is not needed for the SRAM since it is fast enough to allow single-cycle access, but the flash can only be accessed up to 20 MHz in a single cycle. For higher frequencies up to 60 MHz which is the maximum operating frequency of the LPC2138 additional wait state cycles are introduced. The LPC2138 contains a unit named Memory Acceleration Module (MAM) that reduces this delay using a combination of caching and prefetching. The effects of the MAM are similar to a cache. However, there is also a bug caused by the MAM implementation of the LPC213x-series [NXP07], which might corrupt data read if a data fetch is initiated from SRAM while code is running on the on-chip flash. There are two workarounds for this issue which can be found in the errata sheet. Running the application from the SRAM avoids the MAM problem and lets the application run without the effects of a cache.

There are two physical connections between the host and the target system, both utilizing the Olimex® ARM-USB-OCD programming device. The first connection uses the JTAG interface and is used to control the target microcontroller and to download data into the flash. The second connection uses the USB-to-Serial converter which is contained in the programming device to transfer the compiled application which shall be measured as well as the test data from the host to the target and the measurement results back from the target to the host. The exact communication process is described in the following section.

The host system on which the analysis for all tests were performed is a PC using an Intel™ Core® 2 Duo CPU (T7200) running at 2GHz with a memory of 2GB running under a Debian GNU/Linux system with Kernel revision 2.6.27.6. This system was also used for the development of the WCET analysis prototype.

5.1.4 Target Software Layout and Host-Target Communication

When powering up, the target system starts at `0x00000000`. The operation usually located at this address is `ldr PC, Reset_Addr` which will jump to the reset handler located in the flash. The reset handler sets up the stack and different operation modes as defined in [Mar06], where also a more detailed description of the boot process can be found. After the basic set up the reset handler jumps to the main function which contains the user generated code.

The main function sets up the UART communication and sends a command prompt to the host using the serial communication line. When the host receives the prompt, it sends the compiled application which shall be measured to the target. The target stores it at `0x40000200` which is at the start of the usable SRAM since the memory range from `0x40000000` to `0x400001ff` is reserved for the Philips boot loader. When the application has been completely received it is sent back to the host which can verify the transmission. If there has been a transmission failure the host resends the application until the failure is corrected. This part of the target application is called the *loader* since it is used to download the second part of the framework, the *caller*, and the target frame to the SRAM of the LPC2138.

After the verification the host signals the target to start the application. However, this is not directly the software under test but a small stub, the *caller*, which receives the test data and writes it on the variables used by the target frame, then executes the target frame for a single test data set and sends the measured execution-time back to the host. This part of the target software is referred to as *caller* since it calls the target frame. The host can send a new test data set to perform measurements or quit the caller and return to the loader. For the program an executeable binary is compiled which contains the *caller*, as well as the instrumented target application code. The *caller* function which is automatically generated by the WCET analysis framework to set the variables for the code under test is described in section 4.15.1. In contrast to the *MoDECS* prototype multiple measurements can be taken at once, therefore only one version of the target code is necessary. In the *MoDECS* implementation a separately instrumented target frame was needed for the instrumentation of each PS.

The time basis for the measurements is the timer T0 which is set to run at the CPU clock frequency. Before a frame is executed, the timer is restarted using the `PREPARE_MEASUREMENT` macro. Figure 5.4 shows the code used to perform the instrumentations. Each time `IP_MEASUREMENT` is called, the value of the counter `T0TC` which is located at `0xE0004008` is read, and the obtained value is written to `0x40000044 + [0x40000040]` where 224 bytes of unused RAM are located. The memory holds up to 55 measurement values and one pointer to the next free space

```
1  #define MSTART ((volatile uint32_t *) 0x40000040)
2
3  #define PREPARE_MEASUREMENT  \
4    *MSTART = MSTART;       \
5    T0TCR = 0x00000002;        \
6    T0TCR = 0x00000001;
7
8  #define IP_MEASUREMENT                      \
9  asm volatile (                              \
10 "        push    {r2,r3}           \n\t" \
11 "        mov     r2, #1073741824   \n\t" \
12 "        ldr     r3, [r2, #64]     \n\t" \
13 "        add     r3, r3, #1        \n\t" \
14 "        str     r3, [r2, #64]     \n\t" \
15 "        mov     r2, #-536870912   \n\t" \
16 "        add     r2, r2, #16384    \n\t" \
17 "        ldr     r2, [r2, #16]     \n\t" \
18 "        str     r2, [r3]          \n\t" \
19 "        pop     {r2,r3}           \n\t" \
20 :                                        \
21 :                                        \
22 : "cc") ;
```

Figure 5.4: Measurement Instructions

which is located at `0x40000040` and increased every time a measurement has been taken. Basically the assembler code in figure 5.4 is the C statement `*(++dest) = T0TC;`. It is written in assembler for two reasons: First, the compiler cannot optimize it. Therefore it is always the same sequence with the same execution-time. Second, it does not alter the register state, which the compiler can detect by the empty clobbered registers list. It is important that the list of clobbered registers is empty to prevent the compiler from changing the code generation. It is not entirely empty since the processor status register is affected by the instrumentation. Both used registers, *r2* and *r3* are secured on the stack and restored when the measurement is completed. When setting two instrumentation points directly in series the difference between both counter values is 26 clock cycles. This value is used to calibrate the measurement process. It is subtracted by the target from each execution-time transmitted back to the host.

An important task is to make the *loader* and the *caller* collaborate. In the *loader* it is easy to specify the location of *caller* by using its address in the prototype declaration

```
int (*\_caller)(void) = (void*)0x40000200;
```

but there is no easy way to get the linker to place a function at an exact memory location. As a second problem the caller uses some serial I/O functions defined by the loader to conserve SRAM space and needs to know where these functions are located in the flash. The first problem requires an individual linker command file. In [Mar06] it is described where to place different segments of the application when compiling for the execution from flash or SRAM. Unfortunately the specific chapters in the book

only explain the process using the graphical interface which is rather expensive and runs on Microsoft Windows® only. To perform the same task using the GNU linker *ld* requires a command file as shown in figure 5.5.

```
1  ENTRY(caller)                               /* specify the entry point         */
2
3  MEMORY                                      /* specify the LPC2138 memory areas */
4  {
5    flash      : ORIGIN = 0x00000000, LENGTH = 512K   /* FLASH ROM               */
6    ram_low    : ORIGIN = 0x40000040, LENGTH = 223    /* free memory             */
7    ram_isp_l  : ORIGIN = 0x40000120, LENGTH = 223    /* Philips ISP bootloader */
8    ram        : ORIGIN = 0x40000200, LENGTH = 32224  /* free RAM area           */
9    ram_isp_h  : ORIGIN = 0x40007FE0, LENGTH = 32     /* Philips ISP bootloader */
10 }
11
12 _stack_end = 0x40007EDC; /* define a global symbol _stack_end                  */
13
14 SECTIONS                 /* define the output sections                         */
15 {
16   . = 0;                 /* set location counter to address zero               */
17   .t0 :                  /* this goes to the start of the SRAM                 */
18   {
19     *(.text_0)
20   } >ram                 /* put all the above into SRAM                        */
21   .text :                /* collect remaining SRAM sections                    */
22   {
23     *(.text)             /* all .text sections (code)                          */
24     *(.rodata)           /* all .rodata sections (constants, strings, etc.)    */
25   } >ram                 /* put all the above into SRAM                        */
26   .data :                /* collect all initialized .data for SRAM             */
27   {
28     *(.data)             /* all .data sections                                 */
29   } > ram                /* put all the above into SRAM                        */
30   .bss :                 /* collect all initialized .data sections for SRAM    */
31   {
32     *(.bss)              /* all .bss sections                                  */
33   } > ram                /* put all the above into SRAM                        */
34   _end = .;              /* end of application SRAM                            */
35 }
```

Figure 5.5: Linker Command File for *Caller*

The solution to guarantee that *caller* is always placed at `0x40000200` is to create a new text section which contains only *caller* and use the *__attribute__* extension in the function definition like shown in the line below:

```
int caller (void) attribute((section (".text_0")));
```

This instructs the compiler to place *caller* in the segment *text_0* which is the first segment placed in the SRAM starting at `0x40000200`. It is important to generate a separate segment for *caller*. Placing the `text_0` segment at the beginning of the `.text` segment will not work since the linker will likely place variables before functions.

The calls to functions created by the *loader* part of the software can be resolved by adding `-Rloader.out` to the command line arguments when linking the *caller*. This causes the linker to scan the object file `loader.out` for exported symbols and allows the *caller* to refer to these symbols and to use the declared functions and variables provided by the *loader*.

The test data are also transferred over the serial connection to the *my_frame* function, which is individually generated for each application and writes the individual values which are located in an unstructured memory area to the global and static variables. The global variables are automatically substituted by global variables so that they behave like static variables but can be written from *my_frame*. This is done by the *tdata* class described in section 4.15.1. After all variables have been initialized the target instrumented application is executed and *my_frame* gives the control back to *caller* which transfers the measurement results back to the host.

5.2 Description and Goals of Tests Scenarios

The selected test scenarios are primarily used to evaluate features that are unique to the new execution-time measurement prototype V3. Features that have been evaluated with the predecessor version V2 are only covered coarsely.

The first series of tests focuses on the processing of loops and is divided into three parts. The first part examines how well loops are handled in general and which types of loops are supported. The second part examines how the deactivation of optimized loop handling techniques affects the processing of loops. The third part examines, how a special handler for a type of loop where exactly one iteration forces a different path would affect the loop performance. As this loop handler is not implemented in the analysis tool yet, this part uses the framework to measure the different paths through the loop but the quantitative estimate of the effect are done by manual calculation.

The second test series compares inlining and black-boxing, the two methods provided for function handling. Not only the WCET results but also the analysis time is compared. These tests showed that it is important to provide both methods since not all functions can be inlined and not all functions can be black-boxed.

The third series of tests examine how the control-flow paths inside expressions which were neglected in the *MoDECS* project influence the analysis time as well as the quality of the calculated WCET bound. The experiments conducted in section 5.5 show that the expected effect can be observed but is not as strong as anticipated.

A further interesting area is to investigate the correlation between the size of program segments and the required number of measurements. This has already been elaborated by Wenzel [Wen06]. Based on his results a path bound of 100 has been selected for all tests described in this chapter.

5.3 Loops

The tests in this section cover different aspects of the loop handling mechanisms provided in the newly developed WCET analysis prototype.

5.3.1 General Loops

The main difference between the *MoDECS* WCET analysis prototype and the new version presented within this work is the ability to analyze loops. The tests performed in this section compare the ability of the presented WCET analysis prototype (V3) to the prototype developed during the *MoDECS* project (V2). The test cases "ADCKonv" and "AktuatorSysCtrl" are also included even if they do not contain loops in order to present a full overview of the WCET analysis tool performance for all test cases. Table 5.2 shows the analysis results for all test cases. The first column shows the name of the test case. Columns V2 and V3 indicate if the test case works using V2 respectively V3. The next column shows the calculated WCET. The overall model checking time, which includes the time to find loop bounds and the time to generate test data, is displayed in column T_{MC}. The last column explains why the test case could not be analyzed. About half of the Mälardalen test cases could not be analyzed because they contain nested loops. This is not per se a limitation of the analysis method but the analysis of nested loops using the proposed analysis method requires a vast implementation effort. This is caused by a weak design decision when imlementing the dtree class and is described in section 4.12. The second frequent reason for failure were floating point variables in control-flow decisions. All tests in table 5.2 have been made using black-boxing unless otherwise noted in the footnotes.

5.3.2 General vs. Specialized Loop Handling

Loop handling is based on the structure of the loop. The goal of the tests performed in this section is to compare the special loop handling methods for single path const iteration and single path variable iteration loops to the generic loop handling method by disabling the special loop handling techniques. Most of the functions have only a single execution path when they are called from *main* using a fixed function parameter. This is not typical or realistic for normal applications but widely used in the Mälardalen benchmarks. All measurements have been performed on the *main* function using function inlining. Loops, which do not fit in any category for specialized loop handling are not included in the results shown in table 5.3, because the goal in this section is to compare specialized and generic loop handling.

The first column in table 5.3 shows the name of the test case. The WCET is shown in the second column. Since there are no test results for the ARM architecure with the GNU compiler and the same compiler switches available, it is impossible to make assumptions about the tightness of the analysis. As mentioned above the WCET would likely differ amongst the individual loop handling techniques for complex target

Testcase	**V2**	**V3**	**WCET [cyc]**	**T_{MC} [s]**	**Reason for Failure**
adpcm	n	n			nested loop
bs	n	y	313	132	
bsort100	n	n			nested loop
cnt	n	n			nested loop
compress	n	n			unstructured loop with goto
cover	n	y	5749	0	
crc	n	y[a]	113858	129	
duff	n	n			nested loop
edn	n	n			nested loop
expint	n	n			nested loop
fac	n	n			recursive functions
fdct	n	y	9886	0	
fft1	n	n			nested loop
fibcall	n	y	1250	32	
fir	n	n			nested loop
insertsort	n	n			nested loop
janne_complex	n	n			nested loop
jfdctint	n	y	12144	0	
lcdnum	n	y[b]	601	3	
lms	n	n			uses float in branch decision
ludcmp	n	n			nested loop
matmult	n	n			nested loop
minver	n	n			nested loop
ndes	n	n			[c]
ns	n	n			nested loop
nsichneu	n	y	646	73000	
prime	n	y	368305	24	
qsort-exam	n	n			nested loop
qurt	n	n			uses float in branch decision
recursion	n	n			uses recursion
select	n	n			nested loop
sqrt	n	n			uses float in branch decision
statemate	n	y	1916	1284	
AktuatorMotorregler	y[d]	y	4142	1182	
ADCKonv	y	y	574	134	[e]
AktuatorSysCtrl	y	y	82	27	[e]
$\sum$ Testcases: 35	3	10		~21h	

[a] requires inlining (icrc) and black-boxing (icrc1)
[b] analysis does not consider the effect of volatile
[c] cannot be inlined (→ nested loop) nor black-boxed (free function argument pointer)
[d] loops were resolved manually
[e] test case does not contain loops but is included for a complete overview of test cases

Table 5.2: Analyzeable Test Cases in V2 and V3

Testcase	WCET [cyc]	SP/CI Paths meas	SP/VI T_{MC} [s]	MP/VI Paths meas	Paths MC	Paths inf	T_{MC} [s]
bs	313	4	2	14	4	10	132
cover	5749	3	73[a]	14801[b]	0	14797	24238
fdct	9886	5	36[c]	19	0	0	36
fibcall	1250	3	12	86	29	27	186
jfdctint	12144	4	32[d]	81	0	0	32

[a]32, 25, and 16 for loops in functions swi120, swi50 and swi10
[b]14400, 3000 and 100 in functions swi120, swi50 and swi10 + 1 path in main
[c]18+18 in function fdct
[d]8 in function main, 12+12 in function jpeg_fdct_islow

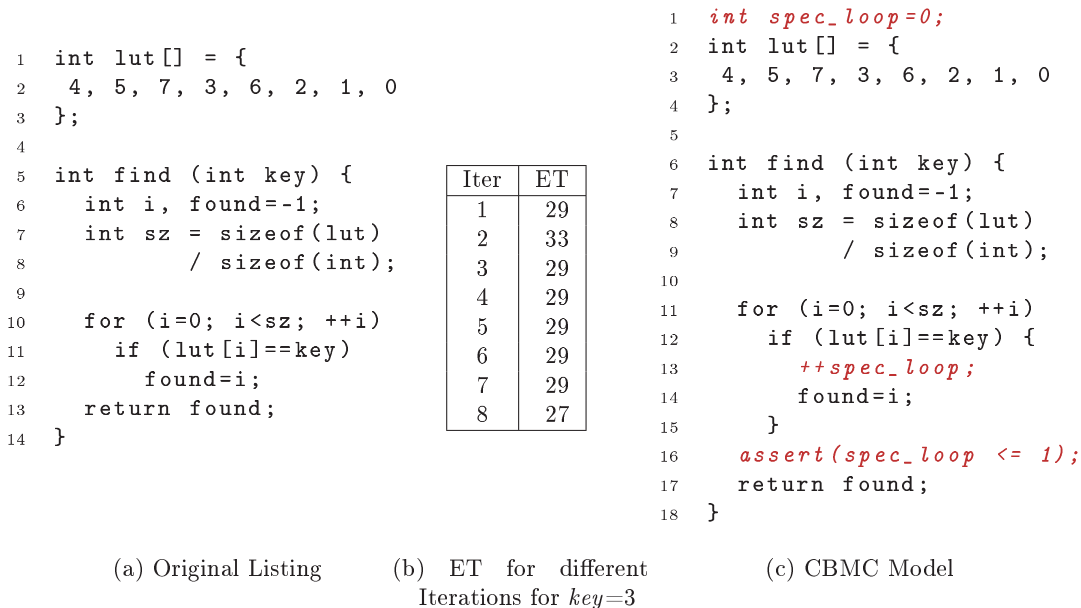
Table 5.3: Generic and Spezialized Loop Handling

architectures. The third column shows the number of paths that have to be measured for the single-path constant-iteration (*SP/CI*) loop type. Since there is only a single path, which is always covered by the random-generated test data the time required for model checking is always 0 for this loop type. For all of the shown test cases there is only a single end-to-end path. Since this path spans across multiple program segments it is measured for each PS individually. For all test cases there are no paths which require model checking or are infeasible using the SP/CI loop type. The next column shows the test results for the single-path variable-iteration (*SP/VI*) loop type. For the number of paths which need to be measured the same applies as for the single-path constant-iteration loop type. The main difference between SP/CI and SP/VI is that the loop bound is unknown in the second case and has to be determined using model checking. The application is then executed with the generated test data so that the maximum loop iterations are actually executed. The time T_{MC} which is displayed in this column is the time required to find the loop bound using model checking. The next four columns show the results for the generic loop approach which covers multiple-path variable-iteration (*MP/VI*) loop type. For this loop type the longest path for each iteration is searched. The sum of these per-iteration WCET paths is the WCET of the whole loop. The drawback of this approach is that at least one instrumentation point per iteration, which means in the loop body, is required. This can introduce a high instrumentation overhead for other hardware architectures, especially for all architectures using a memory cache. The first column shows the number of paths that have to be measured to cover each path inside the loop for each iteration. The test cases *bs* and *fibcall* introduce some control-flow inside the loop header. The paths introduced by the expressions in the header can partly be reached. This is shown in the column *Paths MC*, which gives the number of paths discovered using model checking. The majority of paths is infeasible and their number is shown in the column named *Paths inf*. The last column, T_{MC} shows the time required to cover all paths or mark them as infeasible using model checking. On the *ARM*® architecture the special loop handling algorithms only speed up analysis but on other architectures they might as well tighten the WCET

bound since fewer instrumentation points are required for the special loop handling algorithms.

5.3.3 Reduced Overestimation for 1:n-Loops

The term 1:n-loops is used to describe loops that execute the same path for each iteration except for a single iteration which behaves differently. A special loop measurement logic was considered but not implemented for this kind of loop because of the limited project time. The results in this section are calculated from manual measurements which have been performed for individual paths of the examples.



```
1  int lut[] = {
2   4, 5, 7, 3, 6, 2, 1, 0
3  };
4
5  int find (int key) {
6    int i, found=-1;
7    int sz = sizeof(lut)
8           / sizeof(int);
9
10   for (i=0; i<sz; ++i)
11     if (lut[i]==key)
12       found=i;
13   return found;
14 }
```

Iter	ET
1	29
2	33
3	29
4	29
5	29
6	29
7	29
8	27

```
1  int spec_loop=0;
2  int lut[] = {
3   4, 5, 7, 3, 6, 2, 1, 0
4  };
5
6  int find (int key) {
7    int i, found=-1;
8    int sz = sizeof(lut)
9           / sizeof(int);
10
11   for (i=0; i<sz; ++i)
12     if (lut[i]==key) {
13       ++spec_loop;
14       found=i;
15     }
16   assert(spec_loop <= 1);
17   return found;
18 }
```

(a) Original Listing (b) ET for different Iterations for *key*=3 (c) CBMC Model

Figure 5.6: 1:n-Loop with ET for Different Iterations

Figure 5.6 is a typical example for a 1:n-loop. If the search *key* matches a value in the *lut* array the index is stored in *found* and returned at the end of the function. This special handling takes four cycles more than any other iteration of the loop. When checking the control-flow of the loop body the data flow analysis identifies *key* as variable and therefore concludes that the control-flow in the loop is data dependent. This assumption, although it is correct, leads to the generic loop handling, searching the maximum execution-time for each iteration. When performing measurements on the loop the result is 11 cycles for the loop header. For the loop body, including the jump to the loop head, the result is 22 cycles if the *if* branch is taken, respectively 18, when it is not taken. Therefore the calculated execution-time of the loop is $\text{WCET}_{\text{genloop}} = 8 * 11 + 7 * \max(22, 18) = 242$ cycles instead of $\text{WCET}_{\text{goptloop}} = 8 * 11 + 6 * 18 + 22 = 218$ cycles. This imprecision can be avoided by restricting the number of executions for the special path. Figure 5.6c shows how this can be accomplished easily using a model checker. Using the model figure 5.6c the model checker needs less than a second to verify the given assertion in line 16.

After the effects of 1:n loops have been demonstrated using a small example the effects shall be examined on the basis of the binary search example *bs* of the Mälardalen benchmarks as shown in figure 5.7. The given example has been altered to favor the worst and average case instead of the best case by checking if the element has been at the end of the loop, a modification we would make when optimizing for minimal WCET. After this modification we can measure a WCET of 262 cycles instead of 282 cycles for the *binary_search* function since it is known that the contents of the key array are constant as well as the value of $key = 2$ it can be concluded that the control-flow within the loop is also constant and needs only the measurement of a single path.

```
1  int binary_search(int x)
2  {
3    int fvalue, mid, up, low ;
4
5    low = 0;
6    up = 14;
7    fvalue = -1 /* all data are positive */ ;
8    while (low <= up) {
9      mid = (low + up) >> 1;
10     if ( data[mid].key > x )  {
11       up = mid - 1;
12     } else if ( data[mid].key < x ) {
13       low = mid + 1;
14     } else {
15       up = low - 1;
16       fvalue = data[mid].value;
17     }
18   }
19 }
```

Figure 5.7: A complex 1:n Loop in the Binary Search Test Case

Consider the more interesting case when $key = unknown$. In this case it cannot be assumed that the control-flow in the loop body is constant, instead the generic approach has to be used, calculating the loop bound and searching the longest execution-time for each iteration of the loop. Since the exit criteria might match at each iteration, the longest path for each iteration is the path marked in cursive font in figure 5.7. The WCET for the whole loop the would be calculated as $\text{WCET}_{\text{loop}} = |\text{Num.Iterations}| * \text{WCET}(\text{Iteration})$. What the control logic of the generic approach does not (yet) know, is that the longest path can be executed only once since afterwards the loop is exited. Measuring the individual paths within the loop we get 10 cycles for the loop header, 33 cycles for the shortest path, 49 for the intermediate path and 63 for the longest path. By applying the same procedure as shown in figure 5.6 it can be determined that the longest path can only be executed once. However, since it remains unknown, how often the remaining paths can be executed, the path with the highest execution-time is chosen from the remaining paths.

As last step, the WCET of the whole loop is calculated. Using the generic approach and a maximum of three iterations we get an execution-time of

$WCET_{genloop} = 4 * 10 + 3 * \max(33, 49, 63) = 229$ cycles

instead of

$WCET_{goptloop} = 4 * 10 + 2 * \max(33, 49) + 63 = 201$ cycles.

5.4 Function Inlining vs. Black-Boxing

The V3 WCET calculation prototype offers two types for function handling, inlining and black-boxing. Inlining means that the whole function is expanded in the code during the path analysis and test data generation while black-boxing means that the WCET of the function is determined independently from the calling function and inserted in an individual PS which is created for the function call. Inlining offers the benefit that the context from which the function is called (i.e. possible value ranges for arguments) can be considered but increases the complexity of the analyzed application. On the other hand, black-boxing hides the complexity of the function but might overestimate the execution-time of the function in a specific context. Figure 5.8 demonstrates how the calling environment can influence the execution-time of the called function.

```
1  const int lut_x[] = {
2     1, 2, 4, 8, 16, 0
3  };
4
5  int lut(int value)
6  {
7    const int *xptr = lut_x;
8    int index = 0;
9    while (value > *xptr
10           && *xptr) {
11     xptr++;
12     index++;
13   }
14   if(! *xptr)
15     index=-1;
16   return index;
17 }
18
19 int my_main ()
20 {
21    return lut(3);
22 }
```

(a) Listing

int value	ET [cyc]
1	68
2	101
4	134
8	167
16	200
32	240

(b) lut Execution Times

Figure 5.8: Inlining vs. Black-Boxing

Test Case/Function	FC	PS	Paths EE	Paths meas	Paths rnd	Paths MC	Paths inf	Time MC [s]	WCET [cyc]
bs									
binary_search	0	3	5	14	4	0	10	132	282
B: main	1	3	1	3	3	0	0	0	313
I: main	0	4	5	4	4	0	0	0	312
cover									
swi120	0	1	122	1	1	0	0	0	3766
swi50	0	1	61	1	1	0	0	0	1596
swi10	0	1	12	1	1	0	0	0	356
B: main	3	5	1	5	5	0	0	0	5749
I: main	0	3	89304	3	3	0	0	0	5749
crc									
icrc1	0	3	3	17	12	4	1	5	404
B: icrc	0	6	cannot be black-boxed: unbounded loop						
I: main	1	12	256	28	2	24	2	132	113858
fdct									
B: main	1	3	cannot be black-boxed: pointer in function call						
I: main	0	5	4	5	5	0	0	0	9886
fibcall									
fib	0	3	6	6	2	1	3	3	1219
B: main	1	3	1	3	3	0	0	0	1250
I: main	0	3	6	3	3	0	0	0	1250
jfdctint									
jpeg_fdct_islow	0	4	4	2	2	0	0	0	12114
B: main	1	3	2	1	1	0	0	0	12144
I: main	0	6	4	4	4	0	0	0	12144

Table 5.4 “Comparison between Inlining and Black-Boxing” is continued on the next page →

Test Case/Function	FC	PS	Paths EE	Paths meas	Paths rnd	Paths MC	Paths inf	Time MC [s]	WCET [cyc]
lcdnum									
num_to_lcd	0	1	16	16	2	1	13	3	56
B: main	1	3	3	3	3	0	0	0	601
I: main	0	1	16	170	16	0	154	0	601
prime									
divides	0	1	1	1	1	0	0	0	1678
even	1	1	1	1	1	0	0	0	936
prime	2	2	unbounded loop						
swap	0	1	1	1	1	0	0	0	64
B: main	2	n.a.	cannot be black-boxed: prime is unbounded						
I: main	0	1	45	45	1	0	44	0	368305
statemate									
interface	0	3	5.6e5	268	26	34	208	121	263
init	0	1	1	1	1	0	0	0	143
generic_KINDERSICHERUNG_CTRL	0	6	8472	142	34	3	105	165	75
generic_FH_TUERMODUL_CTRL	0	9	4.2e7	682	276	5	401	502	201
generic_EINKLEMMSCHUTZ_CTRL	0	1	27	27	1	0	26	14	71
generic_BLOCK_ERKENNUNG_CTRL	0	3	124	86	15	9	62	82	71
FH_DU	4	11	84065	502	95	1	406	382	1480
B: main	3	1	1	1	1	0	0	0	1916
I: main	0	21	4.3e26	1832	246	68	1518	12042	1843
AktuatorMotorregler									
Tab1DS0I2T2084_AktMotReg	0	3	8	38	16	10	12	8	674
B: AktuatorMotorregler	3	1.6e18	712	712	52	22	638	1174	4142
I: AktuatorMotorregler	0	3.2e20	767	767	71	18	678	1524	4052

Table 5.4: Comparison between Inlining and Black-Boxing

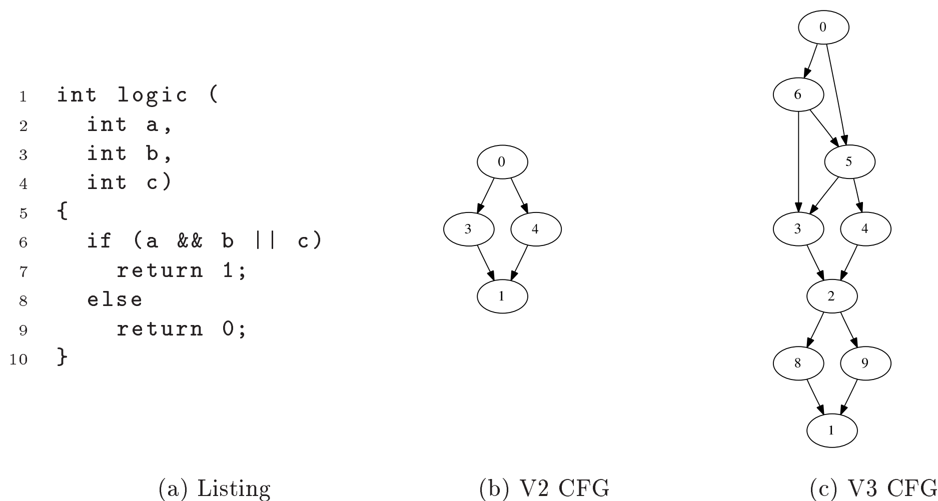
Figure 5.8a shows the listing of a short look-up table function, which uses an array of integers to calculate $\lfloor ld_2(value) \rfloor$ for the range 0 to 64. However, the main function does not use the whole input range, in fact it uses a constant value. Figure 5.8b shows the execution-time of `lut` for different values. It can be seen that the execution-time for the input values 2, which is 101 cycles, and 32, which is 240 cycles and equal to the WCET of the function, differ by 139 cycles. The WCET of the `main` function includes an additional calling overhead of 30 cycles. Thus we get a WCET estimate of 131 cycles when using inlining versus an estimation of 270 when using black-boxing.

It has been shown in the previous example that there can be a big overestimation when using black-boxing. Table 5.4 shows results from a selection of the Mälardalen benchmarks as well as on industrial application (the other two case studies do not contain function calls).

The first column contains the name of the test case. The indented lines contain the names of analyzed functions within each test bench. The lines prefixed with "B:" and "I:" represent the main function of the application and the analysis results for black-boxing respectively inlining. Functions called from other functions have been analyzed using black-boxing (this is only the case in *statemate*). The next column shows the number of function calls in the analyzed function followed by the number of program segments "PS". The next column titled "Paths EE" shows the number of end-to-end paths. The paths that need to be measured are shown in the next column described as "Paths meas". Due to paths in loop iterations this number may be greater than the number of program segments. "Paths rnd", "Paths MC" and "Paths inf" give the number of paths covered by random test data and model checking, respectively the number of infeasible paths. The next column "Time MC" shows the time required for model checking, which includes the time for loop bound analysis and the time for test data generation. Finally the last column "WCET" gives the calculated estimate for the WCET in cycles.

5.5 Control Flow in Expressions

In this section the different handling of control-flow paths inside expressions in the V2 prototype and the V3 prototype is examined. Control flow in expressions arises from C short-circuiting in logical AND (&&) and OR (||) expressions as well as from the use of the conditional expression operator (?:) as described in section 2.1.2. The effect can be observed best using a small example like the one shown in figure 5.9a. The generated CFG using V2 is shown in figure 5.9b and the CFG generated for V3 is shown in figure 5.9c. Since V2 ignores the CFG inside expressions, the resulting CFG is simpler than the CFG of the V3 prototype and consists only of a representation of the *if*-statement in BB 0, the *if*-branch in BB 3, the *else*-branch in BB 4 and the function exit in BB 1. The CFG built using V3 also includes the *if*-statement in BB 0, the *if*-branch in BB 8, the *else*-branch in BB 9 and the function exit in BB 1. Additionally it includes the CFG nodes 3, 4, 5, 6, and 2, which do not represent basic



(a) Listing (b) V2 CFG (c) V3 CFG

Figure 5.9: Handling of Expression Paths in V2 and V3

blocks but are used to model the control-flow within the logic expression as described in section 4.6.

The results of the WCET analysis using V2 and V3 are shown in table 5.5. The same colors for V2-1/V3-1 and for V2-2/V3-5 indicate corresponding test cases. It can be seen that the WCET case V3-2 has not been reached and that the WCET has been underestimated using the V2 prototype. In fact, V3-1, V3-2 and V3-4 which all have different control-flows are all represented by V2-1 and V3-3 and V3-5 are represented by V2-2. When using V2 it is decided by the random-generated test data which actual control-flow path is chosen through the logic expression when measuring it. As logic expressions are commonly used in (industrial) C applications this can lead to underestimation of these applications, which is a serious problem. The possible underestimation increases with the complexity of the logical expression. As there are always the same structures used to build complex logic expressions we can expect an underestimation of 2-3 cycles per shortcut taken, except when function calls in logic expressions are used. In this case, the observed effect is likely to be much higher. However, they have not been observed in any test case, except in prime.c from the Mälardalen Benchmark suite where the expressions `return ((prime(x) && prime(y)))` can be found at the end of the main function. V3 does not support function calls in expression paths. The Black-Box function approach does not work for these cases since it requires instrumentation points placed before and after the function call, which is not possible in this case.

Table 5.6 shows the results when applied to a subset of the test cases. Black-boxing was used to resolve function calls. The differences, which are likely caused by the underestimation of expression paths, are small but measureable and undetectable using V2. An other source for different WCET results is the different segmentation

Nr		Path	a	b	c	ET[cyc]
V2	1	0,3,1	-2767	-6	194	66
	2	0,4,1	0	6659	0	67
V3	1	0,6, 3,2,8,1	3356	-31	-9	66
	2	0,6,5,3,2,8,1	12	0	75321	70
	3	0,6,5,4,2,9,1	-2	0	0	69
	4	0, 5,3,2,8,1	0	0	7328	67
	5	0, 5,4,2,9,1	0	23	0	67
	6	0,6, 3,2,9,1	-	-	-	n.a.
	7	0,6,5,3,2,9,1	-	-	-	n.a.
	8	0,6,5,4,2,8,1	-	-	-	n.a.
	9	0, 5,3,2,9,1	-	-	-	n.a.
	10	0, 5,4,2,8,1	-	-	-	n.a.

Table 5.5: Measurement Results for the Expression Path Example

which moves PS borders and may hide some infeasible paths but make others detectable. In large test cases it is not always possible to identify the source of different WCET estimates.

Testcase	V2				V3			
	Paths EE	PS	Paths Meas	WCET [cyc]	Paths EE	PS	Paths Meas	WCET [cyc]
nsichneu	1.2e+106	62	1308	618	1.7e+120	73	5103	646
statemate	8.5e+5	28	586	1522	4.2e+7	35	1709	1510
AktuatorMotorregler	1.9e+11	14	234	4136	1.6e+18	21	712	4142
ADCKonv	144	2	74	572	2102	4	102	574
AktuatorSysCtrl	97	1	97	82	388	5	51	82

Table 5.6: Effects of Control Flow in Expressions in Industrial Case Studies

5.6 Results

This section presents a discussion of the results from the test runs described in the previous three sections. Significant results are explained and it is also discussed why the observed effect was only very weak in some cases.

5.6.1 Loops

It can be seen from table 5.2 that about half of the tests contain nested loops and cannot be analyzed with the presented WCET analysis tool. This is not a limitation

of the proposed analysis method, but a result of a bad design decision during the prototype implementation which is described in section 4.12. However, one third of the test cases can be analyzed and due to the used loop analysis techniques the model checking effort is low.

It is shown by the different analysis times for the specialized and the generic loop handling techniques in table 5.3 that the analysis performance benefits strongly from the specialized loop handling techniques. What cannot be seen on the *ARM*® architecture is that this benefits not only the analysis performance but also the quality which is in this case the tightness of the resulting WCET bound. This can be assumed based on the fact that less instrumentation points (inside loops) are required whe using the specialized loop handling techniques. Additional instrumentation points have to be inserted in the loop body for the generic loop analysis method, each requiring to flush the cache and the pipeline, for complex hardware architectures. In this case the analysis overhead for small loops which would normally fit completely into the cache can expected to be huge.

The effect of the 1:n-loop optimization described in section 5.3.3 is also considerable but it could not be directly observed in the used test benches. The analysis in this section was performed manually because due to the limited project time the analysis framework does not have support for this loop analysis optimization.

5.6.2 Function Inlining vs. Black-Boxing

The conducted tests compare two strategies to resolve function calls. The first strategy, inlining, replaces the function call with the code in the called function, creating local variables as necessary and works similar to C++ inline functions. However, this inlining is only virtual to calculate control-flow paths and program segments. When performing measurements a function call is performed like usual. The second strategy is called black-boxing. The idea is to perform WCET analysis for individual functions. When a function is called it is simply treated as a black-box with unknown internal structure but a known WCET time.

Both analysis methods have advantages and disadvantages. Inlining generates a high degree of complexity and analysis effort and is beneficial for small functions while black-boxing reduces the complexity and analysis effort but can lead to overestimations. The largest observed overestimation was in *statemate* with 3.96%. In the same test case the model checking time was increased from 1284s for black-boxing to 12060s using function inlining. When analyzing single path applications even an opposite effect can be observed like in *fibcall* where the model checking time is 32s with black-boxing and 0s with inlining. A look at figure 5.10 explains why inlining works better for this test case than black-boxing. When analyzing *fib(int n)* n is a free variable. However, the single-path constant-iteration loop (SP/CI) handling method can only be used when there are no data dependent control-flow decisions within the loop body and the number of iterations is constant. Since the number of iterations depends on the unbound variable n the single-path variable-iteration

```
1  int fib(int n)
2  {
3      int  i, Fnew, Fold, temp, ans;
4
5      Fnew = 1;  Fold = 0;
6      for ( i = 2; i <= 30 && i <= n; i++ )
7      {
8          temp = Fnew;
9          Fnew = Fnew + Fold;
10         Fold = temp;
11     }
12     ans = Fnew;
13     return ans;
14 }
15
16 int main()
17 {
18     int a;
19
20     a = 30;
21     fib(a);
22     return a;
23 }
```

Figure 5.10: Listing of *fibcall* Case Study

(SP/VI) loop handling has to be used. Model-checking has to be used to calculate the loop bound and the test data to reach it. In some cases a loop contained in a called function might even be unbounded when no function parameter is given and black-boxing will fail. The penalty for free function parameters is even higher if the loop body has multiple paths. The generic loop handling algorithm checks each path through the loop body for each iteration, which leads to 14400, 3000, and 100 paths which have to be analyzed for the *swi120*, *swi50* and *swi10* functions in the *cover* test case, requiring a total model checking time of 14797 seconds.

5.6.3 Control Flow in Expressions

The last series of tests covered control-flow in logical AND (&&) and OR (||) expressions as well as the conditional expression operator (? :). The consideration of C-shortcircuit paths for logical expressions can increase the number of end-to-end paths dramatically. A taken shortcircuit causes an underestimation of the WCET. In most of the test cases except for *statemate* the WCET is a few cycles higher when taking the control-flow inside expressions into account. The problem is that the consideration of control-flow paths can change the segmentation of the application. When PS are changed, this can lead to cases where new infeasible paths are detected because mutually exclusive paths are moved from separated segments into the same PS. The opposite, that mutually exclusive paths are moved from the same PS to different PS, allowing them to be executed at the same program run, might also

happen. The reduced WCET bound in *statemate* might be caused by one of the described effects.

It is important to note that the analysis of expression paths is safe with the V3 prototype because all paths that are created by the logical expression are coverd by measurements. Due to the movement of PS borders the calculated WCET estimate of the V3 prototype can be lower than estimate of the V2 prototype as it can be seen in table 5.6. This artifact is caused by infeasible paths but the result from the V3 prototype implementastion is still safe.

5.7 Summary

In this chapter the test environment, both hard- and software has been described. The test cases have been introduced and the test results for various test scenarios have been shown to test different effects when analyzing loops, function calls and control-flow in expressions. The chapter has been concluded by a short discussion of the results from the individual measurements.

Chapter 6

Related Work

This chapter gives an overview of current work related to WCET analysis and other topics relevant to this work like WCET oriented programming, model checking, loop and cache analysis and timing anomalies.

When a application is described in one of the following sections, the authors of the application and their affiliation as well as an URL of the application's web presence is listed. However, if a project cannot be assigned to one or more persons or their affiliation is unknown, this information is omitted.

6.1 WCET Analysis

In this section different static and dynamic or hybrid WCET analysis projects are discussed. Some of the presented projects are only at their beginning stage and are doing basic research, others have already emerged into commercial products. Major parts of the overview are from an ACM survey article by Wilhelm et al. [WEE$^+$08] where a more detailed description of some of the tools can be found. Some additions and updates to this survey article have been made to reflect new tools and the changes to the existing tools made since the publication of the article. Especially all participants of the WCET tool challenge 2008 [HGB$^+$08] have been included in this section. The described analysis tools are categorized in static WCET analysis, where no execution of the code is involved, and measurement-based or hybrid analysis where the examined application is executed either on the target hardware or a simulation of the target hardware. A more complete classification of WCET analysis tools can be found in [KP05] where a classification of tools based on the representation level on which the tools operate, the analysis of flow facts and the execution-time modelling is introduced. At the end of this section there is an overview table which lists the target platforms supported by the individual WCET analysis tools.

6.1.1 Static WCET Analysis

This section gives an overview of current WCET analysis tools implementing static WCET analysis. This list might not be complete but it should include all major projects. AiT developed by AbsInt Angewandte Informatik GmbH and Bound-T developed by Tidorum Ltd. are commercial products. All other tools are prototypes from academic research projects.

AiT

Originator: AbsInt Angewandte Informatik, Saarbrücken, Germany and Reinhard Wilhelm, Programming Languages and Compiler Construction, Saarland University, Germany
URL: `http://www.absint.com/ait/`
AbsInt is a spin-off company of the Saarland University. Research on the WCET analysis tool is done at AbsInt, which makes the tool available commercially, and at the Saarland University. AbsInt's AiT timing analysis tool operates on object code level and obtains upper bounds for execution-times of code fragments which are usually functions of an application. The advantage of performing the analysis on object code is that all registers and memory locations are known and a very precise static analysis is possible. Information that cannot be extracted from the executable file can be provided by the user, either using a separate parameter file or using annotations in the source code. The tool uses line number information which is contained in the object file to assign the source annotations to specific locations in the executable. The user annotations may not only contain information like loop bounds and flow facts but also the value of registers and variables. This feature is used to check applications which can run in multiple different modes depending on the value of a register (i.e. startup, diagnostics mode, or normal operation). The analysis starts by reconstructing an annotated control-flow graph (CFG) from the object code. The next step is the value analysis using *abstract interpretation* (AI) [FW99] and, based on that, the loop bound analysis and the pointer analysis. The cache analysis which is also based on abstract interpretation uses the memory addresses found in the value analysis and categorizes memory access in sure hits and possible misses. The next analysis step is pipeline analysis which gives an upper bound for the execution-time of each basic block. The last step, the bound calculation, uses the local WCET of the basic blocks to determine the WCET path using ILP.

In cases where the automatic loop bound calculation and the jump analysis for indirect jumps and branches does not work user annotations are required. Additionally the tool relies on the architecture specific calling conventions.

calc_wcet_167

Originator: Raimund Kirner and Peter Puschner, Real Time Systems group at Vienna University of Technology, Austria

URL: `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`
calc_wcet_167 reads WCETC source code, which is written in a subset of ANSI-C with annotation extensions for loop bounds and infeasible paths [Kir02]. The tool works with a modified version of a compiler which generates object code and additional information for the WCET analyzer. The tool can be integrated within the Matlab/Simulink toolchain [KLFP02] where it modifies the code generation to include annotations in the generated code. The compiler keeps track of control-flow information during code transformations so timing analysis for highly optimized code can be performed [KP03, Kir03]. The execution-times are calculated using ILP under consideration of infeasible paths [PS97]. The results can be imported into Matlab/Simulink models via dedicated WCET blocks.

The tool requires manual annotations and therefore the analysis quality depends on the quality of the annotations (especially for infeasible paths).

Bound-T

Originator: Tidorum Ldt., Helsinki, Finland
URL: `http://www.tidorum.fi/bound-t/`
Bound-T was developed by State Space Finland under contract with the European Space Agency (ESA) and further developed by Tidorum Ldt. The tool uses an application binary with included debugging information to determine an upper bound for the execution-time of a function, including all called sub-functions, as well as an upper bound for the stack usage. Loop bounds are extracted from typical software-generated loop patterns or provided as code annotations, which have to be written in a separate text file and not in the source code. The annotations refer to the program via named entities (labels, subroutines) or structural properties (loops, calls). Like AiT, Bound-T is independent of the programming language since it uses compiled code. Bound-T has general facilities for modeling control-flow and integer arithmetic, but requires specialized processor models for each supported processor. From the object code an annotated CFG is extracted where the edges contain the WCET for the basic block they represent. Counter-based loops are modelled using Presburger arithmetic as a set of equations or inequalities. To bound loop iterations, Bound-T tries to identify loop counter variables, which are modified by a finite value on each iteration. When Bound-T identifies the initial and final value of the loop counter variable, the number of iterations can easily be calculated. The generated equation set is analyzed and solved using the omega calculator from Maryland University [Pug91]. The WCET is calculated using implicit path enumeration techniques (IPET) using the *lp_ solve* tool [Ber97] by M. Berkelaar. The result of the computation is an upper WCET bound in an easy parsable text file and graph files showing call graphs and annotated CFG graphs in the DOT [GN99] format.

As a requirement for the analysis the analyzed functions must not be recursive and the control-flow must be reducible. Dynamic jumps are analyzed based on commonly used compiler patterns but not all constructs are supported. The aliasing analysis

(pointer analysis) is only very basic. Loop bound analysis does not handle multiplication and bit-operations (and, or, not, shift, ...). Like AiT Bound-T depends on the architecture specific calling conventions.

Chalmers University Prototype

Originator: High-Performance Computer Architecture Group, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden
URL: `http://www.ce.chalmers.se/research/group/hpcag/project/wcet.html`
The tool analyzes Power-PC binaries and derives safe upper bounds for a subset of the Power-PC instruction-set. The tool integrates path and timing analysis in a cycle-accurate hardware-model which is extended to handle unknown input values [Lun02]. Before the simulation is started, all data flow and control-flow information that can be statically derived from the application, which is all input-independent information, is analyzed. When the simulator encounters an input-dependent branch decision, both paths are executed, leading to the well known path explosion problem. Therefore the control-flow is merged again. How often the control-flow is merged again is a trade-off between analysis effort and analysis quality. Timing anomalies are avoided by applying code transformation techniques. The cache analysis can identify data structures that can be safely cached. This improves the worst-case cache performance. It is also possible to determine the worst-case data-cache performance for predictable data structures when the exact access scheme is known a priori, even if the exact location is statically unknown [Lun02].

Loops with unknown bounds (i.e. input dependent) are not recognized - the user has to provide annotations for them. Path-merging has a negative impact on the analysis quality. To achieve tight WCET bounds long-running simulations should be run at the final stage of the application developed.

Chronos

Originator: Xianfeng Li, Yun Liang, Tulika Mitra and Abhik Roychoudhury, National University of Singapore, Singapore
URL: `http://www.comp.nus.edu.sg/~rpembed/chronos/`
$\chi\rho\acute{o}\nu o\varsigma$ is the ancient Greek god of time and also the name of the WCET analysis tool of the National University of Singapore [LLMR07]. Chronos uses a C-source file and the target processor configuration as input. Based on the source a data-flow analysis is performed to determine the loop bounds of the analyzed application. When the analysis fails the user has to provide annotations. The analysis results can also be improved by user-generated infeasible-path information. The analyzer core operates on the object code of the application. The first analysis step is the reconstruction of the CFG from the object file. Chronos supports out-of-order pipelines, dynamic branch prediction and instruction caches. The analysis determines the upper bound for the execution-time by calculating the cache miss/hit cases and jump predictions based on operations in the previous basic blocks and within the current basic block

[LMR05]. To avoid timing anomalies all possible instruction schedules have to be analyzed. Chronos avoids this by a fixed-point analysis of the time intervals at which instructions enter and leave individual pipeline stages [LRM04]. The instruction cache is modeled using an ILP-based technique [LMW99]. As a last step IPET is used to combine loop bounds and user annotated infeasible paths information.

Chronos does not analyze data caches and the data flow analysis and loop bounds computation are limited. The tool relies on user input to detect infeasible paths.

Heptane

Originator: A. Colin and I. Puaut, Institut de recherche en informatique et systèmes alátoires (IRISA), Rennes, France
URL: `http://www.irisa.fr/aces/work/heptane-demo/heptane.html`
The Heptane (Hades Embedded Processor Timing ANalyzEr) tool analyzes source code on which a tree-based analysis can be performed and/or binary files which can be analyzed using ILP. The tree-based analysis is fast but can produce overestimation while the ILP method requires higher computing power but yields tighter results. For loop bounds Heptane relies on user annotations which support non-linear or even non-rectangular nested loops and use *Maple* and *Maxima*, two computer algebra tools, for the tree-based method and *lp_ solve* and *CPLEX* for the ILP-based analysis. Pipelines are analyzed using an off-line simulation of the instruction flow through the pipeline. Cache effects are examined using an extended approach of Frank Mueller's static cache simulation [Mue00]. Finally a branch prediction based on the cache of recently taken branches is used [CP00]. The pipeline, instruction cache and branch predictor are implemented in a microarchitecture-independent formalism, allowing the whole framework to be easily retargeted to a new architecture [CP01].

The Heptane tool offers no kind of flow analysis. Therefore all loop bounds must be annotated manually. There is also no detection of infeasible or mutually exclusive paths leading to pessimistic upper WCET bounds. The tree-based bound calculation (timing schemata) relies on the compiler keeping the control-flow specified in the syntax tree when creating the control-flow graph like V3 does. The tool supports no data caches.

SPARTA

Originator: David Whalley, Florida State University, USA, and Frank Mueller, North Carolina State University, USA, and Chris Healy, Furman University, USA
URL: `http://moss.csc.ncsu.edu/~mueller/sparta.html`
To perform timing analysis of a single task or function the user compiles the source code with a modified compiler which includes the number of loop iterations, control-flow, and instruction characteristics in the object file. During the compilation loop analysis is performed. The loop analysis can handle non-rectangular nested loops

by summations [HSR+00]. The cache analysis supports direct-mapped and set-associative caches [Mue00] and uses data-flow analysis to categorize the cache behavior of each instruction. For function calls instances are created so that an individual categorization for each instance can be used. The cache categorization is used to analyze pipeline stalls and cache miss delays during the pipeline simulation which obtains the upper bound for each path [HWH95]. The worst-case path is found by an iterative search using a fixed point analysis as stopping criterion [Mue00, AMWH94].

Loop bounds for outer loops in non-rectangular loop nests have to be known statically. They do not have to be known when using parametric timing analysis support. There is no support for pointer analysis or dynamic allocations. Recursion is forbidden. The timing analyzer offers only support for simple architectures like the Atmel® ATMega™ microcontroller. Since the prototype works on entire end-to-end paths the performance ranges from minutes for small applications to hours/days for entire systems.

TuBound

Originator: Albrecht Kadlec and Raimund Kirner, Institute of Computer Engineering, Vienna University of Technology, Austria, and Jens Knoop and Adrian Prantl and Markus Schordan, Institute of Computer Languages, Vienna University of Technology, Austria
URL: `http://costa.tuwien.ac.at/tubound.html`
TuBound [PSK08] is part of the CoSTA (Compiler Support for Timing Analysis) project. The project goal is to create a compiler which supports timing analysis. Flow facts can best be analyzed on the source code level but exact timing analysis requires the object code of the application. A timing analysis supporting compiler can transform flow facts from source code to object code. This way both, flow fact extraction and timing analysis can be performed on the optimal representation level. The compiler can even perform optimizations that change the flow facts as long as it updates the flow annotations in the object code according to these transformations. A WCET aware compiler can produce code that does not allow timing anomalies, which is a second important goal of the project. TuBound is the prototype implementation of an integrated WCET analysis tool and a compiler. The first analysis step is the CGF generation. After the CFG generation follows the interval analysis, which determines possible variable ranges for all program locations. Based on the variables a set of symbolic equations is generated and solved by a term-based loop bounder (TeBo) which is written in Prolog. The WCET calculation tool CALC_WCET_167 uses IPET to calculate the WCET from a CFG whose edges are annotated with the maximum execution-time of the respective basic blocks. Another goal within the CoSTA project is the development of a standardized annotation language which contains the strengths of the currently available WCET annotation languages. At the moment each tool has its own annotation language and performing WCET analysis with multiple tools requires annotations for each individual tool.

TuBound is in an early development phase. However, the project team participated in the WCET tool challenge 2008 conducted by the Real-Time Research Center at Mälardalen University [HGB+08].

wcc

Originator: Design Automation for Embedded Systems Group, Computer Science, Chair 12, TU Dortmund, Germany
URL: `http://ls12-www.cs.tu-dortmund.de/research/activities/wcc/motivation/index.html`
Similar to TuBound *wcc* [PLM08] is not a WCET analysis tool but a WCET aware compiler framework. Wcc uses the *ICD-C* compiler framework to generate a high level intermediate representation (ICD-C IR) for C code and to perform optimizations like dead-code elimination or loop unrolling. The ICD-C IR code is used for loop analysis which uses abstract interpretation (AI), interprocedural program slicing and polyhedral loop evaluation. As a next step, the code selector transfers the ICD-C IR to a low level intermediate representation (LLIR). The flow fact analysis works on both, ICD-C IR and LLIR to calculate loop iteration counts, recursion depths and execution frequencies of instructions, relative to some other instructions. AiT is used to generate the timing model. CRL2 is AiT's internal representation of control-flow. Wcc uses a LLIR to CRL2 translator (LLIR2CRL) to translate the low level representation of the code to AiT, which calculates the timing model for a given section of code and uses a CRL2 to LLIR translator (CRL2LLIR) to translate the results back to wcc. WCC allows to generate optimized code for and to statically analyze arbitrarily distributed program objects within a freely definable memory hierarchy by making information on the memory model available in the optimization stages at the ICD-LLIR level. Additionally wcc performs some WCET targeted optimizations: Procedure Cloning (also known as Function Specialization) is an optimization for functions that are often called with constant arguments. In this case, the arguments can be removed and imported into the called function. Procedure Positioning is a well known compiler optimization with the goal to improve I-cache behavior by reducing the number of cache conflict misses. This can be accomplished by smart placement of functions. Loop Nest Splitting is an optimization technique that can reorder the nesting of loops to improve (WCET) performance. Wcc also supports scratchpad memories and I-Cache locking.

6.1.2 Measurement Based and Hybrid WCET Analysis

This section describes the currently available and developed WCET analysis tools implementing measurement-based analysis and hybrid analysis, which means that the WCET analysis process includes a static and a measurement-based or a simulator-based portion. A separation between these two categories is virtually impossible since each tool includes at least some static analysis steps which would make it

automatically a hybrid approach. In the following section the term measurement-based analysis will be used for tools where there is virtually no static analysis and hybrid analysis for implementations that focus equally on static and measurement-based analysis methods. RapiTime is a commercial product developed at Rapita Systems Ldt., the other tools are research projects.

FORTAS

Originator: Real Time Systems Group at Vienna University of Technology, Austria and Formal Methods in Systems Engineering Group at Technische Universität Darmstadt, Germany
URL: `http://www.fortastic.net/about.html`
The FORTAS project [BT08] is concerned with the analysis of embedded software, especially the timing analysis of control software written in C. FORTAS is designed to provide a tool for testing which is more reliable and predictable than the usual ad hoc testing and faster and more convenient than classical static analysis, which requires detailed knowledge about the target hardware as well as significant human effort. It is planned that the tool does not only cover timing analysis but also does analysis of power consumption and energy uptake. FORTAS uses abstraction methods from software model checking to generate test data for execution-time or power measurements. The measurement results are stored in a database to generate a timing model and an annotated state machine. The tool uses an abstraction refinement loop to achieve the required granularity.

FShell (FORTAS Shell) [HSTV08] provides a frontend for the measured data as well as the model checker and measuring process. It supports interactive use and a rich scripting language. *FShell* reads queries and extracts the results from the measurement database. If there are no sufficient data for a given query the *FShell* causes the analysis framework to perform abstraction refinement, test data generation and measurements so that the query can be answered. The data generated during this process is added to the data base.

Currently the project is at an early stage but some of the tools provided already have a limited functionality.

MoDECS V2

Originator: Ingomar Wenzel, Bernhard Rieder, Raimund Kirner and Peter Puschner, Real Time Systems group at Vienna University of Technology, Austria
URL: `http://www.modecs.cc`
The *MoDECS* V2 prototype [WKRP05] is similar to the prototype presented in this work. It uses model checking for test data generation and execution-time measurements on automatic generated program segments with parametric size to gather timing information. The final WCET calculation is performed by a longest path search or using ILP with *lp_solve* [Ber97, Lps]. Besides the *CBMC* model checker the

MoDECS prototype supports the *SAL* model checker which has completely different semantics than C and is far inferior to *CBMC* when generating counter examples for C applications. Therefore only *CBMC* is supported in the V3 prototype.

The *MoDECS* prototype cannot analyze code containing loops or function calls. Also the control-flow inside logic *and* and *or* expressions and conditional expressions is neglected which makes the WCET estimations highly unreliable for code containing these code constructs.

OTAWA

Originator: TRACES team at IRIT labs, University of Toulouse, France
URL: `http://www.otawa.fr/`
OTAWA [CS06] is a framework of C++ classes dedicated to static analysis of programs in machine code and to the computation of WCET. OTAWA provides state of the art WCET analysis methods like IPET and a lot of facilities to work on binary programs (control-flow graphs, loop detection, ...).

While OTAWA is aimed to be a framework to assist WCET analysis there exist already some tools. The most important is the *RANGE* tool [dMBCS08] which extracts flow facts and detects loop bounds from source code by flow analysis and abstract interpretation. The *ARM*® pipeline analysis uses execution graphs [RS07]. The project participated in the WCET challenge 2008 [HGB+08].

RapiTime

Originator: Rapita Systems Ltd., York, United Kingdom
URL: `http://www.rapitasystems.com/rapitime`
Rapitime is a commercial measurement-based timing analysis and profiling system targeted at medium to large industrial applications. Measurements are taken for a section of code (usually a basic block) using user generated test data or test data generated during the operation of the analyzed system. According to the structure of the application the measurement results are combined to estimate the WCET for the whole application. In addition to the timing information RapiTime generates probability distributions as well as the BCET for the individual sections and the whole application [BCP02, BCP03, BBN05]. The timing information can be collected using a software instrumentation library, a lightweight software instrumentation combined with external measurement hardware or pure hardware instrumentation (like Nexus or ETM). In addition to measurements on the target hardware simulators can be used to gather the timing information. Users can add annotations to guide the measurements (i.e. loop bounds). RapiTime is a spin-off of the University of York where the pWCET tool has been developed.

RapiTime cannot analyze programs with recursion and with non-statically analyzable function pointers. The main difference between V3 and RapiTime is that the segmentation is based on a selectable path bound in V3 and on fixed syntactic rules

in RapiTime. Further RapiTime requires user generated test data which makes the quality of the analysis dependant on the quality of the test data. If the test data are unfavorable there might even be a high underestimation of the WCET. In contrast to RapiTime, V3 does the test data generation by itself and ensures the coverage of each path within all program segments.

SWEET

Originator: Björn Lisper, Programming Languages Group, Mälardalen Research and Technology Centre (MRTC) and Mälardalen University (MDH), Sweden
URL: `http://www.mrtc.mdh.se/index.php?choice=projects&id=0017`
SWEET, the SWEdish Execution-Time Tool, was initially a joint project of Mälardalen University (SE), C-Lab in Paderborn (DE), and Uppsala University (SE) and has now fully moved to Mälardalen University. As a result of the geographical separation of the research facilities SWEET became developed very modular, allowing different analyzers and tools to work rather independently [Gus00, Eng02, Erm03]. Like *TuBound* and *wcc*, *SWEET* is embedded in a research compiler. The flow analysis is performed on the intermediate code (IC) of the compiler after structural optimizations have heen made which ensures that the analyzed structure of the IC is the same as in the object file. First comes program slicing which restricts flow analysis only to those parts of the application which may change the control-flow. A value analysis combined with pattern matching is used to identify constructs like simple loops. Complicated constructs are handled using abstract execution [Gus00] which is a form of symbolic execution based on abstract interpretation. The abstract execution is able to calculate both loop bounds and infeasible-path information automatically. The processor analysis is implemented as a two phase approach. The first phase, the memory access analysis, identifies memory accesses by different instructions and performs an instruction-cache analysis similar to Ferdinand and Wilhelm [FW99] producing execution facts like cache hit/miss predictions or branch prediction outcomes. The second phase, the pipeline analysis, uses the execution facts to perform a cycle-accurate simulation of object code sequences executed on the CPU. For data-dependent instruction latencies the worst-case timing is assumed unless the execution facts specify otherwise. It is possible to use standard CPU simulators as CPU models as long as they are cycle-accurate and can be forced to simulate given code sequences under given preconditions (execution facts) and do not suffer from timing anomalies. To locate timing effects across sequences of two or more basic blocks consecutive simulation runs starting at the same basic block are executed. The resulting WCET estimate can either be computed using a path-based technique, a global IPET technique or a hybrid clustered technique which uses local IPET- and/or path-based calculations. The results are graphically visualized using the DOT application from the GraphViz [GN99] package.

The flow analysis can handle ANSI-C programs including pointers, unstructured code, and recursion as long as the application is compiled with the research compiler. In other cases flow facts can be supplied manually. The hardware analysis does not

handle data caches or out-of-order pipelines or timing anomalies. The path-based WCET calculation requires a well-structured program while the other calculation schemes are not subject to such limitations.

SymTA/P

Originator: Rolf Ernst, Institute of Computer and Network Engineering, Technical University Braunschweig, Germany and Symtavision, Braunschweig, Germany
URL: `http://www.ida.ing.tu-bs.de/forschung/projekte/symtap/`
Symatavision is a commercial spinn-off of the Technical University Braunschweig which offers a wide set of scheduling analysis tools. However, from the internet representation of Symtavision it seems that they no longer offer WCET analysis tools. The SymTA/P tool can still be downloaded from the Institute of Computer and Network Engineering of the Technical University Braunschweig. SymTA/P, which stands for SYMbolic Timing Analysis for Processes, is a hybrid analysis tool which combines platform-independent path analysis and platform-dependent measurements using a cycle-accurate simulator or an evaluation board. The first step is a static analysis to identify basic blocks. Additionally single feasible execution paths (SFP) are introduced. SFPs are series of basic blocks without any input dependent control-flow decision [WEY01, EY97]. An example for a SFP would be a fast Fourier transformation (FFT) or a finite impulse response (FIR) filter. These examples contain loops with nested *if* statements generating numerous control-flow decisions. When performing symbolic analysis, which is the proposed solution in SymTA/P, it can be seen that all the control-flow decisions are in fact data independent and only a single feasible execution path (SFP) exists for the FFT or FIR function and that the FFT or FIR algorithm can be reduced to a single node. After the symbolic analysis all nodes are subject to execution-time measurements. The measurement is the platform-dependent part of the analysis. Measurements can be a simple readout of the internal clock of a simulator or the communication with an in-circuit debugger like Nexus or ETM. For a complete timing analysis input data providing full branch coverage has to be supplied. To avoid the requirement for full path-coverage test data a conservative overhead for pipelining effects, which is the same as starting with an empty pipeline, is added for each node. To consider cache effects the instruction- and data-cache behavior is analyzed [WKE01] and annotated in the according node. The longest path is calculated using IPET where the timing for each node consists of the measured execution-time plus the annotated static cache-access behavior.

The measurements get inaccurate for small basic blocks because of the constant time delays added to cover pipeline effects. This leads to overestimation. Data dependent execution-times are not considered and it is assumed that the user generated test data covers the worst execution-time for individual instructions. Test data covering all branch decisions have to be supplied by the user, as well as the loop bounds for loops with input-dependent loop conditions. Like for all measurement-based timing analysis systems the quality of the applied measurement method has a high impact on the quality of the result.

6.1.3 Overview of Current WCET Analysis Tools

Table 6.1 shows an overview of the WCET analysis tools presented in section 6.1.1 and section 6.1.2.The first column contains the name of the tool followed by the flow analysis method used by the tool. The third column describes the hardware modeling, which can be static analysis, measurement-based or a combination of both methods. The fourth column describes how the final WCET is estimated after the execution-times have been acquired. The fifth column describes the language level on which the tool operates. This can be *S* for C source code (except for RapiTime which accepts C and Ada) or O for object code or both. The last column contains a list of supported target architectures.

6.2 Improving the WCET

Both techniques introduced in this chapter aim at reducing or eliminating input-data dependent control-flow branches. To achieve this, an architecture which supports predicated execution is required. While this feature is not commonly found on modern processors which target the average execution-time, some architectures like the *ARM*® processor series or the Intel® Itanium™ series provide predicated execution. Without predicated execution there is still control-flow generated even if it is camouflaged by the use of conditional expressions. The problem is that it is unknown for the user when predicated code is generated. Modern compilers like the *gcc* compiler offer automatic if-conversion which is actually an essential part of single-path conversion. However, the concept of predicated execution is unfamiliar in C and there exist no ANSI-C language feature to control predication.

6.2.1 WCET-Oriented Programming

The goal of common programming techniques is to reduce the average execution-time at the cost of the best- and worst-case execution-time. The idea behind WCET-oriented programming techniques is that a real-time system has to be designed according to the WCET. The best-case or average execution-time is irrelevant, at least if there is no low priority background task which runs during the idle phase of the real-time tasks. Puschner [Pus03a, Pus03b] defines WCET-oriented programming as follows:

> WCET-oriented programming (i.e., programming that aims at generating code with a good WCET) tries to produce code that is free from input-data dependent control-flow decisions or, if this cannot be completely achieved, restricts operations that are only executed for a subset of the input-data space to a minimum.

Figure 6.1 [Pus05] shows a comparison between traditional average-case oriented programming and WCET-oriented programming. To achieve better performance for

```
1  int binSearch_avg(int key, int a[])
2  {
3    int left = 0, right = SIZE-1, idx, inc;
4    int found = 0;
5    do {
6      idx = (right + left) >> 1;
7      if (a[idx] == key) {
8        found = 1;
9      } else if (a[idx] < key) {
10       left  = idx+1;
11     } else {
12       right = idx-1;
13     }
14   } while (!found && (right >= left));
15   if (found) {
16     return idx;
17   } else {
18     return -1;
19   }
20 }
```

(a) Traditional Programming

```
1  static int binSearch_wcet(int key, int a[])
2  {
3    int left = 0, right = SIZE-1, idx, inc;
4    idx = (right + left) >> 1;
5    for(inc = SIZE; inc > 0; inc = inc >> 1) {
6      right = (key < a[idx] ? idx - 1 : right);
7      left  = (key > a[idx] ? idx + 1 : left);
8      idx   = (right + left) >> 1;
9    }
10   return idx;
11 }
```

(b) WCET-oriented Programming

Figure 6.1: Comparison of Traditional and WCET-Oriented Programming [Pus05]

Tool Name	Flow Analysis	Hardware Modelling	WCET Calculation	Input Level	Architectures
AiT	Value analysis	Static analysis	IPET	O	Motorola PowerPC, Motorola ColdFire, ARM7, HCS12 fam., TMS320C33, C166/ST10, Renesas M32C/85, Infineon TriCore
Bound-T	Linear loop bounds and Ω-Tests	Static analysis	IPET per function	O	Intel-8051, ADSP-21020, ATMEL ERC32, Renesas H8/300, AVR, ATmega, ARM7
calc_wcet_167	user annotations	Static analysis	IPET	S, O	M68000, M68360, C167
Chalmers	–	Modified Simulation	Structure-based	O	PowerPC
Chronos	unspecified loop analysis	Static analysis	IPET	O	SimpleScalar out-of-order processor model with MIPS-like instruction-set architecture (PISA)
FORTAS		Measurements		S	
Heptane	user annotations	Static analysis	Structure-based, IPET	S, O	Pentium1, StrongARM 1110, Hitachi H8/300
MoDECS V2	simple DFA	Measurements	Path-based	S	HCS12 fam., Pentium
OTAWA	AI	execution graphs	IPET	S, O	PowerPC, ARM7
RapiTime	n.a.	Measurements	Structure-based	S, O	Motorola PowerPC family, HCS12 family, ARM, NecV850, MIPS3000
Sparta	unspecified loop analysis	Static analysis	Path-based	O	MicroSPARC I, Intel Pentium, StarCore SC100, Atmel Atmega, PISA/MIPS
SWEET	Value analysis, abstr. exec., synt. analysis	Static anal. for I-caches, pipelinesimulation	Path-based, IPET, clustered	S	ARM9 core, NEC V850E
SymTA/P	Single feasible path analysis	Static anal. for I/D-cache, measurements for segments	IPET	S	Various ARM (RealView Suite), TriCore, i8051, C167
TuBond	term-based loop bounder	Static analysis	IPET	S, O	C167
wcc	AI, Interproc. Progr. Slicing, Polyhedral Loop Eval.	Uses AiT	Uses AiT	S, O	Infineon TriCore 1.3

Table 6.1: Overview of WCET Analysis Tools

the average case (which assumes that there are a few iterations needed to find the key), it would be better to move the check for equality in figure 6.1a line 7 down and check for less and greater keys first. However, it is intuitive for programmers to think first about the loop exit condition before implementing the rest of the loop body. Programmers are used to the design pattern they know and nobody would implement a binary search like figure 6.1b since it is not the way the binary search algorithm is taught or implemented normally.

Algorithm	traditional AVG [cyc]	traditional WCET [cyc]	WCET-oriented AVG [cyc]	WCET-oriented WCET [cyc]
bubble	599	724	609	663
find-first	68	122	103	103
bin-search	94	124	105	106

Table 6.2: Execution time of Traditional and WCET-Oriented Algorithms [Pus05]

Table 6.2 [Pus03a] shows a comparison between traditional and WCET-oriented algorithms. The algorithms tested were bubble sort with an array of 9 elements, find-first (find the first appearance of a key in an array with 10 elements) and bin-search (binary search within an array of 10 elements). The first column lists the test name, followed by the average and worst-case execution-time in CPU cycles for the traditional algorithm and the average and worst-case execution-time for the WCET-oriented approach. It can be seen that the average execution-time of the WCET-oriented implementation is always higher than the average execution-time of the traditional approach. On the other hand, the WCET of the WCET-oriented algorithm is always lower than the WCET of the traditional approach. For real-time systems which always have to be designed for the worst-case execution-time the WCET-oriented programming techniques are better suited. However, there are no libraries providing WCET optimized versions of standard algorithms and even professional code generation tools often generate code which is optimized for the average case.

Additionally, WCET-oriented approaches produce a smaller jitter of execution-times which makes them well suited for control systems. They may also offer a smaller number of (input dependent) control-flow paths or even may have only a single execution path.

6.2.2 Single-Path Conversion

The single-path conversion [Pus02] is a different approach towards the predictability of execution-times. The goal is to write applications which provide only a single input data independent execution path. While the results may look similar to WCET oriented programming, the single-path conversion is applied after the code has been implemented [Pus05]. The single-path conversion uses conversion rules to

convert applications with input-data dependent control-flow into applications with a single execution path. Basically the input dependent branches are executed in sequence and the assignment to a given variable depends on a predicate which is the branch condition. These transformations increase the execution-time and reduce the execution-time jitter. However, support on the target architecture and the compiler is required to create real single-path applications.

```
1  static int binSearch_sp(int key, int a[])
2  {
3    int left = 0, right = SIZE-1, idx, inc;
4    idx = (right + left) >> 1;
5    for(inc = SIZE; inc > 0; inc = inc >> 1) {
6      [key<a[idx]]: right = idx-1;
7      [key>a[idx]]: left  = idx+1;
8      idx = (right + left) >> 1;
9    }
10   return idx;
11 }
```

Figure 6.2: Single Path Programming

Figure 6.2 shows the binary search algorithm from figure 6.1a implemented for single-path programming. The difference to the WCET-oriented approach in figure 6.1a is that the WCET-oriented approach uses conditional expressions to perform only a single assignment. However, the single-path method always calculates the addition or subtraction but depending on the value of the predicate the result is either written to *left* respectively *right* or discarded, with exactly the same timing and the same hardware state. An observer who monitors the hardware at line 9 could not tell if any of the predicated instructions have been performed or not, except for looking at the CPU flags. However, when memory write instructions are predicated there will be a difference in the cache. A nice performance benefit is that there is only a single comparison operation (`key-a[idx]`) needed and the predication can use the *ZERO* and *SIGN* flags to determine `key<a[idx]` and `key>a[idx]`.

6.3 Model Checking

The presented work uses model checking to calculate test data and loop bounds. The following list presents model checkers which were considered to be used for this work and explains why the decision was made in favor of *CBMC*.

Model checking refers the analysis of a given property within a model which can be formulated as a finite state machine. The given property can either fail or hold. Some model checkers include the input values which lead to the violation of the given property in their output data. There are different approaches for model checking:

Symbolic model checking avoids ever building the graph for the FSM. The FSM graph is represented indirectly by a formula in propositional logic which is mapped on a binary decision tree (BDD) [McM93].

Bounded model checking uses the FSM and unrolls it for a fixed number of n steps transforming it into a Boolean satisfiability problem (SAT) and using a SAT solver to prove or disprove the properties or assertions given within the model. The number of n can be increased until all possible violations of the model are ruled out.

Abstraction-based model checkers create a simplified copy of the model where the variables are partitioned into a "visible" and an "invisible" subset and try to verify the given properties on the simplified model. The real and the abstract model are Galois connected. This means if we have a model Ψ and an abstraction of the given model Ψ', then we have an abstraction function η and a concretion function θ so that $\eta(\theta(\Psi')) = \Psi'$ and $\theta(\eta(\Psi)) \supseteq \Psi$. Therefore when the abstraction does not satisfy a given property it does not necessarily mean that the same property actually fails in the real model. The counter examples have to be checked against the real model to determine if they are correct. The abstraction-refinement loop [CGL94] works by creating an abstract model and model checking it. If there is a counter example in the abstraction it is checked if it is also a counter example in the real model. If the counter example works on the real model the process is finished. In the other case variables leading to a dead end space are identified and added to the model and a new iteration of the abstraction-refinement loop begins. The worst-case scenario happens when all variables are visible in the abstraction and $\Psi' = \Psi$.

SAL

Originator: Leonardo de Moura et al. Computer Science Laboratory of SRI International, USA
URL: `http://sal.csl.sri.com/`
The Symbolic Analysis Laboratory (SAL) provides a symbolic and a bounded model checker and was used as first model checker in the V2 prototype [WKRP05] implementation. SAL does not support modulo data types. This means when the operation $255 + 1$ is performed on an 8 bit integer in C the result is 0, while SAL produces an overflow when the datatype is defined in the range [0..255] or 256 when the datatype can hold that value. Additionally none of the bit operators like *and*, *or*, *xor*, *not* and *shift* is supported by SAL. SAL does not support C datatypes and arithmetics since C operates on modulo sets (i.e. a 8 bit character generates a modulo 256 set). Additionally SAL requires a complicated model generation and performs badly for the generated models [WRKP05]. Therefore SAL is not included in the V3 tool version.

Blast

Originator: The BLAST 2.0 Team - Dirk Beyer (SFU), Thomas A. Henzinger (EPFL), Ranjit Jhala (UCSD), and Rupak Majumdar (UCLA), USA

URL: `http://mtc.epfl.ch/software-tools/blast/`
BLAST, the Berkeley Lazy Abstraction Software Verification Tool, uses an abstraction-based method [HJMS02] to perform model checking of C applications which are the models used for input. Due to the abstraction-based algorithm BLAST is very fast and has been used to check large programs [HJMS03, BCH+04, BHJM05, BHJM07] for dead code, memory safety and other (potential) errors. However, at the beginning of the implementation of the V3 analysis tool BLAST provided no way for including the input variables in the counter-example traces and could therefore not be used for test data generation.

CBMC

Originator: Daniel Kroening, ETH Zurich, Switzerland and Edmund Clarke, Computer Science Department, Carnegie Mellon University, United Kingdom
URL: `http://www.cprover.org/cbmc/`
CBMC is a Bounded Model Checker for ANSI-C and C++ programs [CKL04]. The basic verification is to perform loop unwinding and pass the results to a SAT solver. The list of supported language features includes all basic types including float types, all integer operations including bit operations and basic float operations, type casts, side effects, function calls, control-flow statements (goto, return, break, continue, switch), arrays, structs, unions, pointers (deferencing, arithmetic, relational operators, pointer type casts and pointers to functions), as well as dynamic memory (malloc and free functions) [CKL, CK06]. The almost complete support of all ANSI-C features makes it possible to perform model checking on virtually all applications.

The basic function of the tool is to transform the application to static single assignment form (SSA), extract the bit-vector equations and use a SAT solver to calculate the results [CKL04] like shown in figure 6.3.

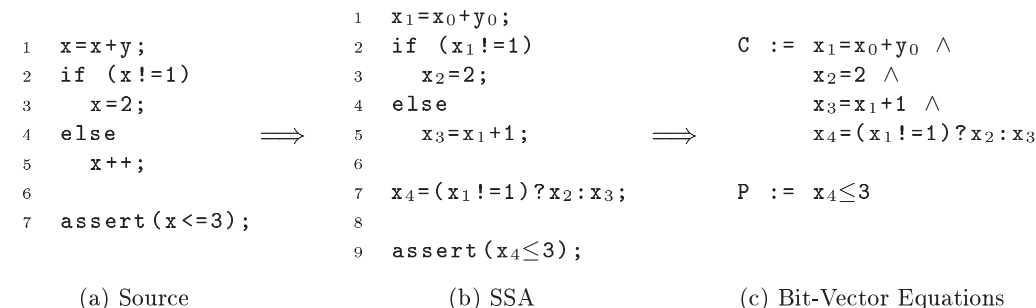


Figure 6.3: Extraction of Bit-Vector Equations with CBMC [CKL04]

CBMC was chosen as a model checker for the new developed V3 timing analysis prototype for multiple reasons: First of all it supports ANSI-C directly and no complicated code transformations are necessary and second, virtually the complete ANSI-C language standard is supported. Additionally CBMC offers superior perfor-

mance and last but not least it is available for free under a relatively open license [CK].

6.4 Program Analysis

This section discusses different static analysis methods used in the WCET analysis tools presented in section 6.1.1 and section 6.1.2 to calculate loop bounds and infeasible paths.

Loop Bound Analysis

Loop bounds can be derived using Presburger arithmetics [HS02], abstract interpretation [The04, GEL05, GESL06], symbolic execution [LS99a], model checking [RPW08], specialized data flow analyzes [CM07], and syntactical analysis on parse-trees [FW99, HSR+00]. Each method has individual drawbacks and advantages. Syntactical analysis for instance is very fast but can only detect specific loop patterns. This is in case of object code based implementations often limited to an individual compiler version. However it is usually very fast. Presburger analysis is also very fast but limited to basic mathematic operators for iteration variables and works only for integer-based iteration variables. Symbolic execution and abstract interpretation are both fast but have weaknesses with pointers as loop iteration variables. Model checking covers the widest variety of loop types but is relatively slow. There are also some approaches which depend on user annotations or let the user supply additional annotations.

Infeasible Paths

To tighten WCET analysis only feasible paths should be considered in the WCET path. The application of value dependent constraints which detect iteration-based constraints as well is described in [HW99]. A method based on partially-known variables, which is a special form of variable range analysis, is used in [APT00] to identify infeasible paths. Abstract interpretation which contains also a variable range analysis is used by [GESL06, GEL06] to identify loop bounds and infeasible paths using conflicting pairs of basic blocks. A similar approach [CMRS05] is also creating pairs of conflicting basic blocks. Conflicts are detected by simple assignments followed by a branch, i.e. `x=5;` conflicts with the following `if (i==0) {...}`. The detection works only for statements of the form *`variable = constant;`* and *`variable relational-operator constant.`* A combination of symbolic execution and path enumeration is used in [KS06] to find loop bounds and infeasible paths at the same time.

6.5 Cache Analysis

Caches come not only in different sizes but also with completely different architectural characteristics. Basic attributes of a cache are whether the cache is used for instructions, data or unified, the mapping (direct or n-way set-associative) of the cache and the replacement strategies. The most frequently used replacement techniques are last recently used (LRU), pseudo LRU, and round robin (RR). In addition to the first level cache (L1) there might be a L2 and sometimes even a L3 cache.

Since cache analysis relies on the cache architecture it can only be performed for a specific processor design or a family of processors having the same cache properties. In order to analyze the caching behavior for a series of memory-access operations the exact memory locations of the individual data items have to be known, which is only the case in object code. While the difference between cache hits and misses are quite large [WM05] the most problematic property of cache is the cache state at the beginning of the analysis. It has long been assumed that the worst-case is an empty cache which causes in a cache miss, but [RWT+06] shows that caches can be a source of timing anomalies.

Abstract Interpretation

Most approaches now use flow-analysis based on abstract interpretation (AI) [CC77, AFMW96]. AI is used to calculate invariants for a specific program point. Different execution paths lead to different invariants; therefore multiple invariants which are specific for a set of execution paths, the calling context, can be calculated. The invariants hold static knowledge about cache, pipeline and branch prediction logic contents and can be used for cache analysis. The cache access is classified in *always hit*, *always miss* and *unclassified* [TFW00]. For each memory access the cache has to be updated with an update function and at control-flow joins the cache contents are merged with the merge function. The merge and update function both depend on the cache organization and replacement strategy. For set associative caches using LRU the data-flow-based approach works well but the predictability drops for other replacement strategies like pseudo-round robin (ColdFire MCF processor family) or pseudo-LRU (PowerPC 7xx family) which are often used in hardware because they are easy to implement and provide high performance.

Implicit Path Enumeration Techniques (IPET)

Another possible implementation of cache analysis is to include the cache analysis in implicit path enumeration (IPET) techniques [LMW95, KS07]. The IPET-based cache analysis uses Cache Conflict Graphs (CCG), one for each conflict. A conflict occurs when two or more basic blocks map different memory locations to the same cache line. The problem with this method is that the IPET constraints and therefore the ILP problems get much bigger through introducing the CCGs in the program

model as noted in [Wil03]. For now the method works for small applications but it is currently far away from practical application for industrial size applications.

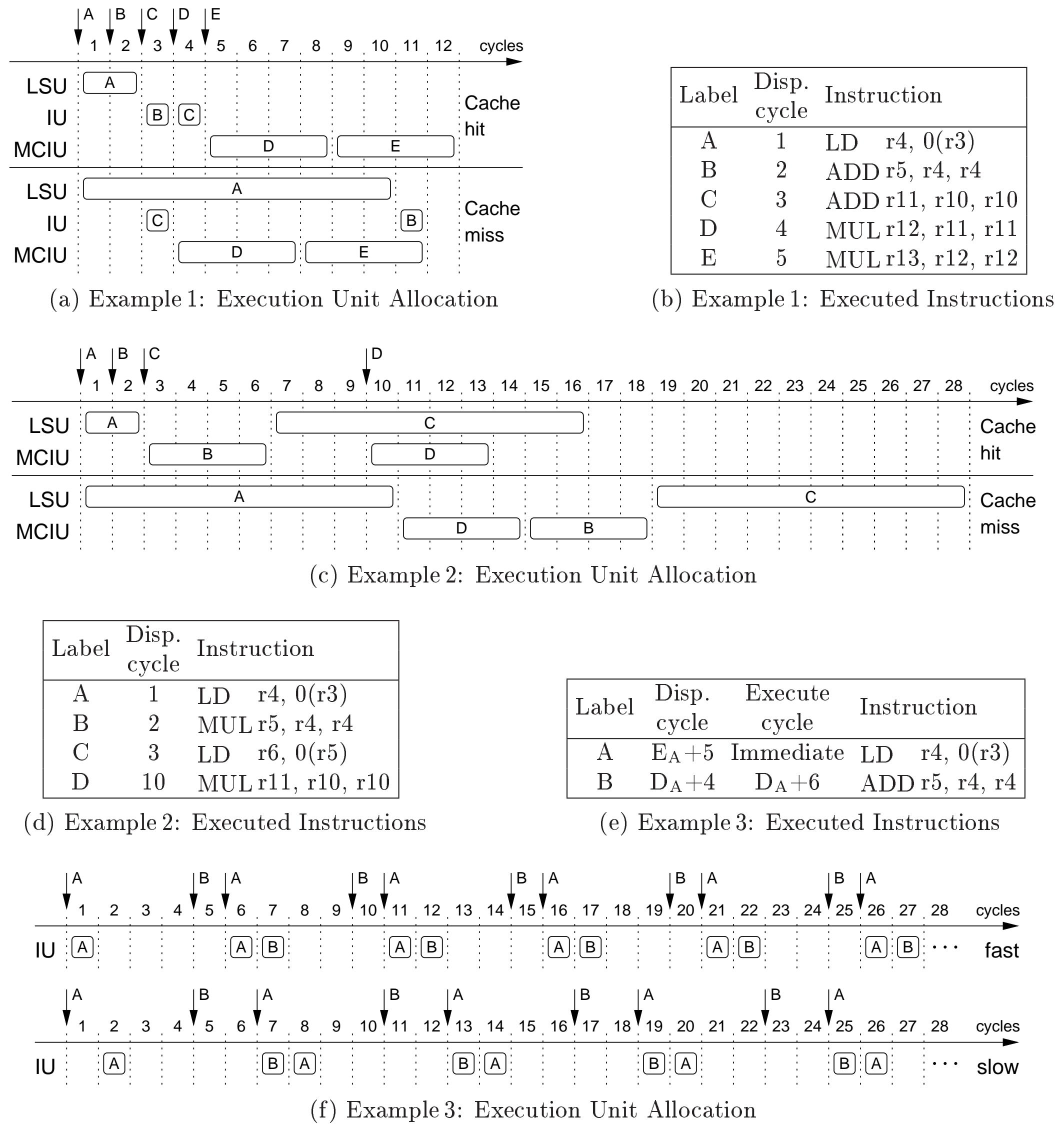
Model Checking (MC)

Model checking has been used in [Met04] for WCET analysis including instruction cache analysis. The methods for cache access and replacement modelling and the merge functions have been implemented similarly to cache analysis methods based on abstract interpretation. The proposed method has the advantage that it can consider concrete execution paths during model checking which reduces the overestimations that are sometimes made by AI or IPET and produces good results at increased analysis cost. In fact the results shown in [Met04] are slightly tighter than results gained from AI.

6.6 Timing Anomalies

Timing anomalies describe unexpected behavior of a local reduction of execution-time that causes a globally increased execution-time or vice versa which can be observed on modern architectures. Term "Timing anomalies" has first been introduced by [LS99b] using a simplified PowerPC architecture without floating point units. The architecture uses a multiple-issue pipeline which can dispatch two instructions each clock cycle as well as separate instruction and data caches. Each functional unit has two reservation stations, which hold already dispatched functions until their operands become available, allowing out-of-order execution of instructions. To avoid unnecessary data hazards register renaming is used. Results are written back in-order by means of a completion unit with a reorder buffer updating the register file from the renaming buffers. However the register renaming unit and completion unit are not required to show the appearance of timing anomalies. The functional units include an integer unit (IU), a multiple-cycle integer unit (MCIU), and a load/store unit (LSU). All modelled resources are in-order resources except the IU and the MCIU, which makes timing anomalies as shown in Figure 6.4 [LS99b] possible.

Figure 6.4a and subfigure b show an example where the cache miss which increases the latency of a single operation by 8 cycles reduces a the latency of a series of five instructions by one cycle. The slower execution of the first command results in a more favorable dispatching of the instructions on the different functional units. An opposite effect is shown in figure 6.4c and subfigure d where the latency is also increased by 8 cycles but the global effect is an increased latency of 12 cycles. In this case the observed delay is much higher expected. The delayed first instruction causes a bad distribution of operations among the functional units of the processor. A very interesting domino effect is shown in figure 6.4e and subfigure f. The notation E_A in subfigure f denotes the instance when instruction A is executed and D_A the time when it is dispatched. In this example A should be executed every fifth cycle within a loop with n iterations. B is dispatched four cycles after the dispatchment of A



(a) Example 1: Execution Unit Allocation

Label	Disp. cycle	Instruction
A	1	LD r4, 0(r3)
B	2	ADD r5, r4, r4
C	3	ADD r11, r10, r10
D	4	MUL r12, r11, r11
E	5	MUL r13, r12, r12

(b) Example 1: Executed Instructions

(c) Example 2: Execution Unit Allocation

Label	Disp. cycle	Instruction
A	1	LD r4, 0(r3)
B	2	MUL r5, r4, r4
C	3	LD r6, 0(r5)
D	10	MUL r11, r10, r10

(d) Example 2: Executed Instructions

Label	Disp. cycle	Execute cycle	Instruction
A	E_A+5	Immediate	LD r4, 0(r3)
B	D_A+4	D_A+6	ADD r5, r4, r4

(e) Example 3: Executed Instructions

(f) Example 3: Execution Unit Allocation

Figure 6.4: Timing Anomalies Examples from [LS99b]

because of data dependencies. In the fast case A is executed immediately after it has been dispatched and B is executed six cycles later, just after A has been dispatched and executed for the second iteration. In the second case, the first execution of A is delayed for a single cycle maybe because of data dependencies on $r3$. This causes a delay of one cycle in the dispatch operation for the next iteration of A. However, B is executed in the cycle where the second iteration of A is now dispatched and therefore the execution of A has to be delayed for a clock cycle. This is repeated for each iteration. In the bottom line each iteration introduces a delay of one clock cycle and therefore a loop with n iterations results in a delay of n clock cycles.

As Lundqvist noted in [LS99b] timing anomalies cannot occur in processors containing only in-order resources. However, he shows in his thesis [Lun02] that timing

anomalies can also occur in processors containing only in-order resources depending on the cache replacement policy.

Schneider developed an integrated WCET analysis framework for the PowerPC 755 [Sch03]. With this framework the appearance of timing anomalies on real processors have been shown, including the domino effect which was shown by Lundqvist [LS99b] on theoretical processor models before.

Wenzel defines the Resource Allocation Criterion (RAC) [Wen03, WKPR05] which is a necessary but not sufficient condition for the occurance of timing anomalies. As a consequence it is safe to assume that hardware without possible dynamic resource allocation decisions does not suffer the impact of timing anomalies. However, the formal definition of the RAC is too restrictive since in covers only cases where exactly one instruction changes its timing behavior.

A first formal definition of timing anomalies has been made in [RWT+06]. This work classifies timing anomalies into scheduling, speculation and cache timing anomalies and gives a formal definition of timing anomalies. Their formal definition is based on a processor model where timing anomalies occur when a local WCET path is not part of the global WCET path. The formal definition is rather complex making it difficult to integrate in a WCET analysis tool.

A more practical formal definition of timing anomalies is introduced by Kirner et al. [KKP09]. Kirner et al. define a timing relevant system state (TRSS) which covers the whole system including, but not limited to, the state of the processor (including cache), I/O, memory, and external cache. The timing relevant dynamic computer state (TRDCS) is the portion of the *TRSS* which is not given by the system hardware or configuration and therefore dynamic. To reduce the complexity of the analysis series decomposition and parallel decomposition are introduced. The series decomposition is the calculation of the longest execution-time for each control-flow node but instead calculating k output states for k input states a state approximation for the output state is calculated, which is then forwarded as input state to the next node. Parallel decomposition is to calculate the WCET of an instruction sequence in two steps. The TRDCS is partitioned into a state space A and B for two hardware components like the instruction cache and the pipeline which may be overlapping. The timing of component A is analyzed and a state is chosen. Based on the results the state space of B is explored and a cumulative timing is calculated. The result of the analysis is a three dimensional function of the timing behavior of the whole system $T = f(h_{wA}, h_{wb})$ for $h_{wA} \in A$ and $h_{wB} \in B$. Timing anomalies are categorized in weak and strong series timing anomalies (because they challenge series decomposition) and parallel inversion or amplification (because they challenge parallel decomposition). For strong series and parallel inversion timing anomalies an increased latency of a sub-series of instruction or a hardware component causes a decreased latency of the whole series of instructions or the hardware system. For weak series and parallel amplification timing anomalies an increased latency of a sub-sequence or sub-system causes a more increased latency of the whole instruction sequence respectively system.

6.7 Other Publications related to this Work

Tree-Based Timing Schema for Loops

Basically tree- or syntax-based loop analysis works by using a timing schema [PBB04]. Given a basic block i preceding and a basic block k following the loop, as well as a loop head j a simple timing schema for the WCET of a loop would be $C_{i,k} = C_{i,j} \otimes (C_{j,j} \odot n) \otimes C_{j,k}$, where $\otimes$ represents an additive operator and $\odot$ a multiplicative operator. For simple execution-times the operators would be an addition and a multiplication. When using probability-based time distributions like RapiTime [PBB04] these operations would be a convolution and a power operator. This approach is also used to calculate the execution-time of loops in this work. It works good for single path loops or loops which contain paths of similar execution-time. In other cases overestimation may occur.

Comparison of Execution Time Measurement Methods

A good overview and comparison of methods for measurement-based WCET Analysis can be found in [Pet03]. The measurement methods are classified in simulation, light-weight software monitoring, heavy-weight software monitoring, hardware supported software monitoring, (software supported) hardware monitoring. The article explains the different methods and discusses their individual strengths and weaknesses. The measurement technique used in this thesis is, according to [Pet03], a light-weight software monitoring technique.

Are ILP and MC bad as Stand-Alone Tecniques?

In his essay [Wil03] Wilhelm demonizes model checking and ILP as standalone technique and explains why both techniques are bad when used as stand-alone techniques. His main critique is the increasing complexity and the state space explosion of the problems when hardware effects are considered. At the moment he is still right regarding ILP. But he does not consider that resourceful minds always manage to find solutions or workarounds for challenging problems and in fact his postulate is proven wrong for model checking in [Met04]. In his work Metzner uses model checking to calculate the WCET of different small to medium sized case studies using a model which contains a cache abstraction with 128 sets of 4 instructions (each 4 bytes) per line and an associativity of two, resulting in a total of 8 KB. The achieved results are compared to those of [Wil03] and generate tighter WCET bounds in all cases at the cost of a larger but still reasonable analysis time.

6.8 Conclusion

In this chapter other WCET analysis tools have been described and new approaches to WCET-oriented programming have been shown. A short introduction

of different model checkers has demonstrated why CBMC was chosen for the WCET analysis prototype. Furthermore two important elements of static analysis which have also been implemented for the introduced prototype, loop bound analysis and infeasible paths analysis, which is implemented implicitly on the level of individual program segments, have been described. An overview of current cache-analysis techniques has been given and current research about timing anomalies has been presented.

Chapter 7

Conclusion and Outlook

This work has shown in theory and by a prototype implementation that the hybrid WCET analysis approach is not only restricted to simple acyclic applications but can also be extended to support loops and function calls.

7.1 Summary of Measurement-Based WCET Analysis

This section gives a brief roundup of the measurement-based WCET analysis approach which has been extended in this thesis to support loops and function calls as well as control-flow in logic expressions.

The presented measurement-based or hybrid WCET analysis method combines static and dynamic measurement analysis methods. The steps performed in the analysis are:

Static Analysis: During this step all control-flow paths through the application are analyzed. Functions are inlined or replaced by a black box, depending on the command line arguments when executing the analysis tool. During the construction of the CFG the flow information of logic expressions are evaluated and integrated in the CFG. All variables are analyzed whether they are input or state dependent or not. In the next step the result of the variable analysis is used to determine if loop headers or bodies are control-flow dependent. For control-flow dependent loops the loop bound is analyzed using model checking.

Control Flow Graph Partitioning: The segmentation of the CFG reduces the number of paths to analyze so that applications with 10^{30} and more end-to-end paths can be analyzed using a few hundred measurements. The CFG partitioning has to avoid path segment boundaries, where instrumentation points will be placed during the execution-time measurements, in loops and function calls.

Test Data Generation: For each path π within a program segment the execution-time has to be known or it has to be ensured that π is infeasible. Test data is reused from previous runs and generated randomly. For all paths which have not been covered by reused test data and random test data, model checking is used in order to create a counter example holding the necessary test data or prove the infeasibility of the given path.

Execution Time Measurements: After the test data has been generated the execution-time of all paths can be measured. Only the execution-time of infeasible paths cannot be measured. This ensures that the path with the longest execution-time is executed during the measurements.

Final WCET Calculation: The final step is the calculation of the WCET of the whole application. This is done by combining the measured execution-times from individual program segments into a single estimate of the WCET using the structural information from the static analysis and the segmentation step.

7.2 Lessions Learned

The V3 prototype implementation introduces three new features to the measurement-based analysis method:

Analysis of Loops The majority of applications contains loops or nested loops. A WCET analysis tool without proper loop support is of little practical use for the analysis of industrial applications. The results of the loop analysis in section 5.3 show that the loop bound analysis using model checking in combination with the loop handling based on the categorization of loops is a convenient approach for loop handling. It is fast and requires less implementation effort than implementations based on abstract interpretation. It has also been shown that the use of specialized loop handling methods speeds up the analysis considerabely in comparison to the generic loop analysis method.

Analysis of Function Calls The analysis of function calls is an essential requirement for a WCET analysis tool. This work has shown that both methods introduced in this thesis, inlining and black-boxing, have their individual advantages and disadvantages. The increased tightness of the WCET bound was not as high as anticipated. The observed maximum of the effect was a 4% lower WCET estimate. The real benefit of function inlining is that loops which are unbounded in the called function are often bounded by the context of the calling function. This makes functions that cannot be analyzed using black-boxing analyzeable using function inlining.

Analysis of Control Flow in Expressions The analysis of control flow in expressions is important since the WCET is probably underestimated if the control flow is ignored. Even if these underestimation are often small, since the logic expressions are usually simple, they are a serious threat to the safety of the WCET analysis. It has ben shown in section 5.5 that this underestimation occurs in real scenarios. Therefore the control-flow analysis of expressions is an important new feature for a WCET analysis tool.

7.2.1 Gained Experiences

This section contains results and personal insights which are not directly related to the measurement-based analysis method but which the author of this thesis would like to share.

An experience learned from the prototype implementation is the importance of well-considered design decisions. While the analysis method supports nested loops in practice, they are not supported in the prototype due to a bad design decision described in section 4.12. A multiple of the time invested in considering data types and algorithms is gained during the implementation phase of an application.

A further insight gained during the project is, that some hardware optimization techniques can be considered in the analysis but the majority requires detailed knowledge of the hardware and memory locations of variables. When simulating hardware effects on source code level the effort is very high and the uncertainness how the compiler translates a specific code construct still remains. Flow-facts on the other hand can better be analyzed on the source code level than on the object code level. The future of WCET analysis tools will probably be to use of source code and object code representation, which can be well integrated within a compiler framework.

It can also be seen that the interdependencies between multiple analysis methods is very strong and influences the WCET analysis. In section 5.5 it has been shown that the analysis of control flow in expressions changes the segmentation of the application. This changes the set of analyzed paths and has more impact on the WCET estimate than the consideration of control flow paths in the analysis.

7.3 Applications of Hybrid WCET Analysis

The developed WCET analysis works well for simple architectures without cache. It is positioned between fast but insecure execution-time measurements and complex static analysis. A major drawback of the proposed hybrid analysis method is the requirement that the control-flow is not altered by the compiler, which is not possible in some cases. For instance in architectures without barrel shifters a command like `x«y` will almost certainly result in a loop or `x/y` will cause a subroutine call on object code level. Hopefully this gap will be filled by WCET aware compilers like *TuBound* or *wcc* that produce flow information in the generated object file.

For architectures with caches the proposed method can be used to acquire a fast but insecure WCET estimation using light-weight instrumentation, or to use heavy-weight instrumentation and sacrifice performance for the safety of the analysis. The integration of cache and pipeline analysis in the proposed method would also be an interesting challenge.

Another important use of the tool is the automatic generation of test data which may not only be used for WCET analysis but can also be very useful for debugging or profiling.

7.4 Future Work

While this thesis extends the measurement-based WCET analysis approach to work with industrial applications there remain still a few open topics which are an interesting and challenging area of research. Some of them have been mentioned in the previous sections.

The most important task would be to change the *dtree* implementation so that nested loops can be analyzed. The supporting of nested loops the majority of test cases from the Mälardalen benchmarks would be analyzeable. The realization of a wider range of specialized loop-analysis techniques would also be an interesting opportunity to improve the WCET analysis tool. During the measurements it has been observed that the performance of the application is severely reduced without optimizations. An integration of the hybrid WCET analysis method with a WCET-aware compiler would greatly improve the usability of the analysis tool.

The support for function calls in paths with control-flow or multiple (nested) function calls is another topic for further research. The current implementation places instrumentations only between statements. This prevents the analysis of these code constructs. In order to allow function calls anywhere in the code a way to place instrumentation points not only between but also inside statements has to be found.

The consideration of control-flow differences between the application source code and object code, which is very important for an analysis based on the source-code level, can be resolved by the integration of flow information into the compiler and annotating all flow transformations in the generated object file. The integration of hardware optimizations in the analysis process is also a challenging research topic. While the cache analysis could possibly be done at a platform independent level the pipeline analysis requires detailed hardware knowledge and the framework partially loses its platform independence.

When measurement-based analysis methods are used for more complex architectures additional hardware features like caches and pipelines have to be considered. It is relatively easy to consider caches when it is assumed that an empty cache is the worst case. In combination with timing anomalies it has been shown that an empty cache is not always the worst case. A challenging task for further research is to find a

solution for this problem and make measurement-based timing analysis safely usable for complex hardware architectures.

Finally there remains one question: Is it safe to integrate processors from the consumer market into dependable embedded real-time systems? The processors are cheap and offer high performance but the complexity of the designs will always be a possible source for faults, as will the decreasing feature size since it makes processors more susceptible to radiation and electrical surges. It is unthinkable that it is impossible to develop simple yet powerful designs that can be used for safety-critical applications. However, until a switch to safer systems is made, we have to live with the current situation and do our best to create predictable systems from unpredictable hardware.

Bibliography

[AEBER04] Mostafa Abd-El-Barr and Hesham El-Rewini. *Fundamentals of Computer Organization and Architecture*. Wiley-Interscience, 12 2004. ISBN 9-780-471-46741-0.

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Static Analysis Symposium (SAS), LNCS 1145*, pages 52–66. Springer, 1996.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006. ISBN 0-321-48681-1.

[AMWH94] Robert D. Arnold, Frank Mueller, David Whalley, and Marion Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. 15th Real-Time Systems Symposium (RTSS)*, pages 172–181, Brookline, Massachusetts, Dec. 1994.

[APE03] APEX Working Group. *Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface*, 2003.

[App99] Andrew W. Appel. *Modern Compiler Implementation in C*. Press Syndicate of the University of Cambridge, New York, NY, USA, 1999. ISBN 0-521-58390-X.

[APT00] H.A. Aljifri, A. Pons, and M.A. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *Proceeding of the IEEE International Performance, Computing, and Communications Conference*, 430–436, Feb. 2000.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, June 1986. ISBN 0-201-10088-6.

[BB00] Guillem Bernat and Alan Burns. An approach to Symbolic Worst-Case Execution Time Analysis. In *Proc. 25th Workshop on Real-Time programming*, Palma, Spain, May 2000.

[BBN05] Guillem Bernat, Alan Burns, and Martin Newby. *Probabilistic timing analysis: An approach using copulas*, volume 1 of *Journal of Embedded Computing, Real-Time Systems (Euromicro RTS-03)*, pages 179–194. IOS Press, 2005. ISSN1740-4460.

[BCH+04] D. Beyer, A. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS: Static Analysis*, Lecture Notes in Computer Science 3148, pages 2–18. Springer, 2004.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. 23rd Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, Dec. 2002.

[BCP03] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems. Technical Report YCS-2003-353, Department of Computer Science, University of York, 2003.

[Ber97] M Berkelaar. lp solve: a mixed integer linear program solver, 1997. ftp://ftp.es.ele.tue.nl/pub/lp solve.

[BHJM05] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with BLAST. In *FASE: Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science 3442, pages 2–18. Springer, 2005.

[BHJM07] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *Software Tools for Technology Transfer*, 9:505–526, 2007.

[Bli02] Johann Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22:183–227, 2002.

[BT08] Sven Bünte and Michael Tautschnig. A Benchmarking Suite for Measurement-Based WCET Analysis Tools. In *First International Conference on Software Testing, Verification and Validation (ICST)*, Lillehammer, Norway, April 2008. IEEE Computer Society Press.

[C89] American National Standards Institute, X3J11, 1430 Broadway, New York, NY 10018, USA. *American National Standards Programming Language C, ANSI X3.159-1989 (C89).*

[C90] International Organization for Standardization (ISO), WG14, ch. de la Voie-Creuse, Case postale 56, Geneva 20, Switzerland. *International Organization for Standardization ISO 9899:1990 Programming Languages - C (C90).*

[C99] International Organization for Standardization (ISO), WG14, ch. de la Voie-Creuse, Case postale 56, Geneva 20, Switzerland. *International Organization for Standardization ISO 9899:1999 Programming Languages - C (C99).*

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994. ISSN 0164-0925.

[Cha09] Robert N. Charette. This Car Runs on Code, Feb 2009. http://spectrum.ieee.org/feb09/7649.

[CK] Edmund Clarke and Daniel Kroening. CBMC Software License. http://www.cprover.org/cbmc/LICENSE.

[CK06] Edmund Clarke and Daniel Kroening. *ANSI-C Bounded Model Checker User Manual*. Carnegie Mellon University, Pittsburgh, PA 15213, August 2nd 2006. http://www.cprover.org/cbmc/doc/manual.pdf.

[CKL] Edmund Clarke, Daniel Kroening, and Flavio Lerda. CBMC Supported Language Features. http://www.cprover.org/cbmc/language_features.html.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CM07] Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *Proceedings of 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2007.

[CMRS05] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis*, pages 40–43, July 2005.

[Cow88] P.D. Coward. Symbolic execution systems-a review. 3(6):229–239, 1988.

[CP00] Antoine Colin and Isabelle Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *RTS*, 18(2):249–274, May 2000.

[CP01] Antoine Colin and Isabelle Puaut. A Modular and Retargetable Framework for Tree-based WCET Analysis. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherland, June 2001. Technical University of Delft.

[CS06] H Cassé and P Sainrat. OTAWA, a Framework for Experimenting WCET Computations. In *3rd European Congress on Embedded Real-Time Software*, 2006.

[DCC07] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Context switch overheads for Linux on ARM platforms. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 3, New York, NY, USA, 2007. ACM.

[dMBCS08] M. de Michiel, A. Bonenfant, H. Casse, and P. Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Embedded and Real-Time Computing Systems and Applications, 2008*, pages 161–166, Aug. 2008. ISBN: 978-0-7695-3349-0.

[Duf83] Tom Duff. A description of Duff's Device on Wikipedia. http://en.wikipedia.org/wiki/Duff's_device, 1983.

[Eng99] Jakob Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, Atlanta, Georgia, USA, May 1999.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Acta Universitatis Upsaliensis, Uppsala, Sweden, Apr. 2002.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis.* PhD thesis, Acta Universitatis Upsaliensis, Uppsala, 2003.

[ESG+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In

Seventh International Workshop on Worst-Case Execution Time Analysis, (WCET'2007), Pisa, Italy, July 2007.

[EY97] Rolf Ernst and W. Ye. Embedded program timing analysis based on path clustering andarchitecture classification. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 598–604, 1997.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proc. of the 1st International Workshop on Embedded Software (EMSOFT 2001)*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[Fis88] J.A. Fisher. Replacing hardware that thinks (especially about parallelism) with a very smart compiler. *Design and Application of Parallel Digital Processors, 1988., International Specialist Seminar on the*, pages 153–159, Apr 1988.

[Fle] The FlexRay Communications System Specification, The Flexray Consortium. http://www.flexray.com/.

[FMC+07] Christian Ferdinand, Florian Martin, Christoph Cullmann, Marc Schlickling, Ingmar Stein, Stephan Thesing, and Reinhold Heckmann. New Developments in WCET Analysis. In *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *Lecture Notes in Computer Science*, pages 12–52. Springer Berlin / Heidelberg, June 2007. ISBN 978-3-540-71315-9, ISSN 0302-9743.

[FMW97] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. Workshop on Language, Compiler and Tool Support for Real-Time Systems (LCTRTS)*, pages 37–46. ACM, June 1997.

[FMW98] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 1998. Selected for special issue SAS'96.

[FSF09] FSF. GLPK (GNU Linear Programming Kit). WWW, 2009. http://www.gnu.org/software/glpk/.

[FW99] Christian Ferdinand and Reinhard Wilhelm. *Efficient and Precise Cache Behavior Prediction for Real-Time Systems*, volume 17, pages 131–181. Springer Netherlands, November 1999. http://www.springerlink.com/content/v0173k6006513gl1.

[GCC] GCC development team, Free Software Foundation (FSF). *GNU gcc compiler command line options for Optimizations and Deugging.* http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html.

[GE97] Jan Gustafsson and Andreas Ermedahl. Automatic derivation of path and loop annotations inobject-oriented real-time programs. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 257–262, 1997.

[GEL05] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 287–297, Feb 2005. ISSN 1530-1443.

[GEL06] Jan Gustaffson, Andreas Ermedahl, and Björn Lisper. Algorithms for Infeasible Path Calculation. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/portals/WCET06/fulltext_link.php?id=667.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.

[GG69] R. L. Graham and R. L. Grahamt. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math*, 17:416–429, 1969.

[GN99] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and Its Applications. *Software - Practice and Experience*, 30:1203–1233, 1999. http://www.graphviz.org/Documentation/GN99.pdf.

[Gro06] Mälardalen WCET Research Group. Mälardalen WCET Benchmarks, 2006. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[Gus00] Jan Gustafsson. *Analysing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Uppsala University, Uppsala, Sweden, May 2000.

[HAM+99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David Whalley, and Marion G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[HGB+08] Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET Tool Challenge 2008: Report. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2008/1663.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[HJMS03] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *SPIN: Model Checking of Software*, Lecture Notes in Computer Science 2648, pages 235–239. Springer, 2003.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4 edition, 9 2006. ISBN 9-780-123-70490-0.

[HS02] N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2002.

[HSR+00] Christoper A. Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert van Engelen. Supporting Timing Analysis by Automatic Bounding of Loop Iterations. *Real-Time Systems*, pages 121–148, May 2000.

[HSTV08] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, Lecture Notes in Computer Science, Princeton, NJ, USA, July 2008. Springer.

[HW99] Christopher A. Healy and David B. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, pages 79–88. IEEE, June 1999.

[HWH95] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proc. 16th IEEE Real-Time Systems Symposium (RTSS)*, pages 288–297, Los Alamitos, California, 1995. IEEE CS Press.

[Joh75] Stephen C. Johnson. YACC – Yet Another Compiler Compiler. Technical Report 32, Computing Science Research Center, AT&T Bell Laboratories, Murray Hill, 1975.

[JSK+07] Jinkyu Jeong, Euiseong Seo, Dongsung Kim, Jin-Soo Kim, Joonwon Lee, Yung-Joon Jung, Donghwan Kim, and Chong Sang Kanghee Kim. *Software Technologies for Embedded and Ubiquitous Systems*, chapter Transparent and Selective Real-Time Interrupt Services for Performance Improvement, pages 283–292. Springer Berlin / Heidelberg, 2007.

[KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[Kir02] Raimund Kirner. The Programming Language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[Kir03] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.

[KKP+07] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET Analysis: The Annotation Language Challenge. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 2007.

[KKP09] Raimund Kirner, Albrecht Kadlec, and Peter Puschner. Worst-Case Execution Time Analysis for Processors showing Timing Anomalies. Research report, Technische Universität Wien, Institut für Technische Informatik, 01 2009.

[KLFP02] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Proc. 14th Euromicro International Conference on Real-Time Systems*, pages 31–40, June 2002.

[Kop97] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications.* Kluwer, 1997.

[Kop98] Hermann Kopetz. The Time-Triggered Architecture. In *Proc. ISORC'98*, Kyoto, Japan, Apr. 1998.

[KP03] Raimund Kirner and Peter Puschner. Transformation of Meta-Information by Abstract Co-Interpretation. In *Proc. 7th International Workshop on Software and Compilers for Embedded Systems*, pages 298–312, Vienna, Austria, Sep. 2003.

[KP05] Raimund Kirner and Peter Puschner. Classification of WCET Analysis Techniques. In *Proc. 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 190 – 199, May 2005.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2 1978. ISBN 978-0131101630.

[KS06] Djemai Kebbal and Pascal Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/portals/WCET06/fulltext_link.php?id=675.

[KS07] Raimund Kirner and Martin Schoeberl. Modeling the function cache for worst-case execution time analysis. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 471–476, San Diego, California, 2007. ACM Press, New York, NY, USA. isbn = 978-1-59593-627-1,.

[KWRP05] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. UBooks Verlag, Dec. 2005.

[LDS07] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying The Cost of Context Switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.

[Lin] The Local Interconnect Network (LIN) Spezification, The LIN Consortium. http://www.lin-subbus.org/.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. *Chronos: A Timing Analyzer for Embedded Software*, volume Special issue on Experimental Software and Toolkit of *Science of Computer Programming*. 2007.

[LLS+07] Insup Lee, Joseph Y-T. Leung, Sang H. Son, et al. *Handbook of Real-Time and Embedded Systems*. Computer and Information Science Series. Chapman & Hall/CRC, 1 edition, 7 2007. ISBN 1-58488-678-1, 800 pages.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[LM97] Y.-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, Dec 1997.

[LME99] Yau-Tsun Steven Li, Sharad Malik, and Benjamin Ehrenberg. *Performance Analysis of Real-Time Embedded Software*. Springer, 1999. ISBN 0-792-38382-6.

[LMR05] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling Control Speculation for Timing Analysis. *Real-Time Systems Journal*, 29(1):27–58, January 2005.

[LMW95] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 380–387, Nov. 1995.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(3):257–279, 1999. ISSN 1084-4309.

[Lps] lp_solve, planning, project_planning, website. Website. http://lpsolve.sourceforge.net/.

[LRM04] Xianfeng Li, A. Roychoudhury, and T. Mitra. Modeling Out-of-Order Processors for Software Timing Analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 92–103. IEEE Computer Society Washington, DC, USA, 2004. ISBN: 0-7695-2247-5.

[LS98] Thomas Lundqvist and Per Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–15, June 1998.

[LS99a] Thomas Lundqvist and Per Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2/3):183–207, Nov. 1999.

[LS99b] Thomas Lundqvist and Per Stenström. Timing Analysis in Dynamically Scheduled Microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 12–21, Dec. 1999.

[Ltd08] Rapita Systems Ltd. RapiTime Website - Description of WCET analysis methods, 2006-2008. http://www.rapitasystems.com/wcetmethods.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, June 2002.

[Mar06] Trevor Martin. *Introduction to the LPC2000.* HITEX Ldt., Sir William Lyons Road, University Of Warwick Science Park, Coventry, CV4 7EZ, first edition, February 2006. ISBN: 0-9549988-1.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN:0792393805.

[Met04] Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347, 2004.

[MIS96] The Motor Industry Software Reliability Association MISRA. Development Guidelines For Vehicle Based Software: C Sub-set. Draft Version 0.1, Issued only to the MISRA and X-By-Wire Consortia for comment and contributions, Dec. 1996.

[MIS98] The Motor Industry Software Reliability Association MISRA. *Guidelines For The Use Of The C Language In Vehicle Based Software.* MISRA, Apr. 1998.

[MIS04] MISRA The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*, Oct 2004.

[MP02] Kevin J. McGrath and James K. Pickett. Microcode patch device and method for patching microcode using match registers and patch routines. United States Patent 6438664, August 20 2002. Advanced Micro Devices Inc.

[MRL02] Tulika Mitra, Abhik Roychoudhury, and Xiafeng Li. Timing Analysis of Embedded Software for Speculative Processors. In *Proc. 15th ACM International Symposium on System Synthesis*, pages 126–131, Kyoto, Japan, 2002. ACM New York, NY, USA. ISBN:1-58113-576-9.

[MSR02] Langenbach Marc, Thesing Stephan, and Heckmann Reinhold. Pipeline Modeling for Timing Analysis. In *SAS '02: Proceedings of the $9^t h$ International Symposium on Static Analysis*, volume 2477 of *Lecture Notes In Computer Science*, pages 294–309, London, UK, 2002. Springer-Verlag. ISBN 3-540-44235-9.

[Mue00] Frank Mueller. Timing Analysis for Instruction Caches. *Real-Time Systems Journal*, 18(2/3):209–239, May 2000.

[NXP07] NXP Semiconductors (founded by PHILIPS). *LPC2138 Errata Sheet Rev. 1.8*, July 2007. http://www.nxp.com/acrobat_download/erratasheets/ES_LPC2138_1.pdf.

[OLI06] OLIMEX Ltd., Plovdiv, Bulgaria. *OLIMEX ARM-USB-OCD Programming Device Description and Schematic*, 2006. http://www.olimex.com/dev/arm-usb-ocd.html.

[OLI07] OLIMEX Ltd., Plovdiv, Bulgaria. *OLIMEX LPC-H2138 Development Board Description and Schematic Rev. B*, 2007. http://www.olimex.com/dev/lpc-h2138.html.

[OS90] A.J. Offutt and E.J. Seaman. Using symbolic execution to aid automatic test data generation. In *Computer Assurance, 1990. COMPASS '90, 'Systems Integrity, Software Safety and Process Security'., Proceedings of the Fifth Annual Conference on*, pages 12–21, 1990.

[PBB04] Stefan M. Petters, Adam Betts, and Guillem Bernat. A New Timing Schema for WCET Analysis. In *4th Int. Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 39–42, 2004.

[Pet03] Stefan Petters. Comparison of Trace Generation Methods for Measurement Based WCET Analysis. In *Workshop on WCET Analysis*, 2003.

[PF99] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 442–449, 1999.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufman Publishers, 2nd. edition, 1998.

[Phe03] Richard Phelan. Improving ARM Code Density and Performance – New Thumb Extensions to the ARM Architecture. Technical report, ARM Ldt, 2003.

[PHI06] PHILIPS. *LPC213X User Manual Rev. 2*, July 2006.

[PK89] Peter Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[PLM08] Sascha Plazar, Paul Lokuciejewskiand, and Peter Marwedel. A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations. In *Proceedings of The 29th IEEE Real-Time Systems Symposium (RTSS) WiP*, volume 29, Barcelona / Spain, December 2008. http://ls12-www.cs.tu-dortmund.de/publications/papers/2008-rtss.pdf.

[Pro08] The Flex Project. flex: The Fast Lexical Analyzer, 2008. http://flex.sourceforge.net/.

[Pro09] The Open Cores Project. Open Cores Project. WWW, 2009. http://www.opencores.org.

[PS90] C. Park and A.C. Shaw. Experiments with a Program Timing Tool based on a Source-Level Timing Schema. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 72–81, 1990.

[PS93] Peter Puschner and Anton Schedl. A Tool for the Computation of Worst Case Task Execution Times. In *Proc. 5th Euromicro Workshop on Real-Time Systems*, pages 224 – 229, Jun. 1993.

[PS97] Peter Puschner and Anton V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *The Journal of Real-Time Systems*, 13:67–91, 1997.

[PSK08] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2008/1661, ISBN 978-3-939897-10-1, also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3.

[PSR92] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 76–84, New York, NY, USA, 1992. ACM. 0-89791-534-8.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM.

[Pus02] Peter Puschner. Transforming Execution-Time Boundable Code into Temporally Predictable Code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[Pus03a] Peter Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[Pus03b] Peter Puschner. Hard Real-Time Programming is Different. In *Proc. 17th IEEE International Parallel and Distributed Processing Symposium, 11th International Workshop on Parallel and Distributed Real-Time Systems*, page 116, Apr. 2003. invited paper for special session.

[Pus05] P. Puschner. Experiments with WCET-Oriented Programming and the Single-Path Architecture. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 205–210, 2005.

[R+] Dominic Rath et al. *Open On-Chip DEbugger (OpenOCD) Online Documentation*. Free and Open On-Chip Debugging, In-System Programming and Boundary-Scan Testing, http://openocd.berlios.de/web/.

[Rat05] Dominic Rath. Open On-Chip Debugger, Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family. Master's thesis, University of Applied Sciences Augsburg, Department of Computer Science, 2005.

[RB92] RTCA/DO-178B. *Software considerations in airborne systems and equipment certification*, 1992.

[RM05] H. Ramaprasad and F. Mueller. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 148–157, 2005.

[RPW08] Bernhard Rieder, Peter Puschner, and Ingomar Wenzel. Using Model Checking to derive Loop bounds of general Loops within ANSI-C applications for measurement based WCET analysis. In *Proceedings of the Sixth Workshop on Intelligent Solutions in Embedded Systems (WISES'08)*, 2008.

[RS07] Christine Rochange and Pascal Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on High-Performance Embedded Architectures and Compilers*, 2(3):109–128, 2007.

[RWSP07] Bernhard Rieder, Ingomar Wenzel, Klaus Steinhammer, and Peter Puschner. Using a Runtime Measurement Device with Measurement-Based WCET Analysis. In *Proceedings of the International Embedded Systems Symposium 2007*, pages 15–26, Amsterdam, Netherlands, June 2007.

[RWT+06] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. <http://drops.dagstuhl.de/opus/volltexte/2006/671> [date of citation: 2006-01-01].

[SA00] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[Sch03] Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Universität des Saarlandes, Germany, Jun. 2003. ISBN: 3-8322-1594-8.

[SEE01] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4th International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Atlanta, Georgia, USA, Nov. 2001.

[SNC+07] Smruti Sarangi, Satish Narayanasamy, B. Carneal, Abhishek Tiwari, B. Calder, and J. Torrellas. Patching Processor Design Errors with Programmable Hardware. *IEEE Micro*, 27(1):12–25, 2007.

[SP81] Micha Sharir and Amir Pnueli. *Program Flow Analysis: Theory and Application*, chapter 7, Two approaches to interprocedural data flow analysis, pages 189–233. Prentice Hall, 1981.

[Sta04] William Stallings. *Operating Systems*. Number 5th edition. Prentice Hall, New Yersey, USA, 2004. ISBN 0-131-479-547.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Number 2nd edition. Prentice Hall, New Yersey, USA, 2001. ISBN 0-135-881-870.

[Tea09] GCC Development Team. *GCC, the GNU Compiler Collection V4.3.3*. The GNU Project, January 2009. http://gcc.gnu.org/.

[TF98] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *The 19th IEEE Real-Time Systems Symposium*, pages 144–153, 1998.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.

[The02a] Henrik Theiling. *Control Flow Graphs For Real-Time Systems Analysis*. Ph. D. Thesis, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[The02b] Henrik Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, October 2002.

[The04] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, 2004.

[TTP] The TTP Protocols, Real-Time Systems Research Group, Institute of Computer Engineering, Vienna University of Technology. http://www.vmars.tuwien.ac.at/projects/ttp/ttpmain.html.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Muller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1—53, April 2008. http://www.mrtc.mdh.se/index.php?choice=publications&id=1485.

[Wen03] Ingomar Wenzel. Principles of Timing Anomalies in Superscalar Processors. Master's thesis, Technische Universität Wien, Vienna, Austria, 2003.

[Wen06] Ingomar Wenzel. *Measurement-Based Timing Analysis of Superscalar Processors*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[WEY01] Fabian Wolf, Rolf Ernst, and Wei Ye. Path clustering in software timing analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):773–782, 2001.

[Wil03] Reinhard Wilhelm. *Why AI + ILP is good for WCET, but MC is not, nor ILP alone*, volume Verification, Model Checking, and Abstract Interpretation of *Lecture Notes in Computer Science*, pages 309–322. Springer Berlin / Heidelberg, Berlin / Heidelberg, 2003. ISBN 978-3-540-20803-7, http://www.springerlink.com/content/wlv2w75db57yn77q.

[Wir01] Niklaus Wirth. *Embedded Systems and Real-Time Programming*, volume 2211. January 2001.

[WKE01] Fabian Wolf, Judita Kruse, and Rolf Ernst. Segment-Wise Timing and Power Measurement in Software Emulation. In *Proc. IEEE/ACM Design, Automation and Test in Europe Conference, Designers' Forum*, pages 165–169, Munich, Germany, Mar. 2001.

[WKPR05] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of Timing Anomalies in Superscalar Processors. In *Proceedings of the Fifth International Conference on Quality Software (IEEE), Melbourne, Australia*, Sep. 2005.

[WKRP05] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Worst-Case Execution Time Analysis. In *SEUS '05: Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Washington, DC, USA, 2005. IEEE Computer Society.

[WKRP08] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Timing Analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, Oct. 2008.

[WKS+05] Ingomar Wenzel, Raimund Kirner, Martin Schlager, Bernhard Rieder, and Bernhard Huber. Impact of Dependable Software Development Guidelines on Timing Analysis. In *The International Conference on Computer as a Tool (EUROCON)*, volume 1, pages 575–578, Nov 2005.

[WM05] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 600–605 Vol. 1, 2005.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–611, Washington, DC, USA, 2005. IEEE Computer Society.

[ZCW04] Jian Zhang, X. Chen, and Xiaoliang Wang. Path-oriented test data generation using symbolic execution and constraint solving techniques. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 242–250, 2004.

[Zha04] Jian Zhang. Symbolic execution of program paths involving pointer structure variables. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 87–92, 2004.

Index

?:, 8, 64, 109, 113
&&, 8, 64, 109, 113

function
    black-boxing, 106
WCET, 35
worst-case execution-time
    measurement-based analysis, 121

abstract interpretation, 39, 116, 133
    cache analysis, 134
abstract syntax tree, 10
ACET, *see average-case execution-time*
AI, *see abstract interpretation*
ALU, *see arithmetic logic unit*
AND, 8, 64, 109, 113
ANSI-C, 5
ARINC, *see avionics application software standard interface*
arithmetic logic unit, 22
AST, *see abstract syntax tree*
average-case execution-time, 35
avionics application software standard interface, 31

basic block, 11
    number of BBs, 30
BB, *see basic block*
BCET, *see best-case execution-time*
best observed execution-time, 36
best-case execution-time, 35
bit operations, 8
black-boxing, 55
BOET, *see best observed execution-time*
branch prediction, 25
    branch history table, 26
    buffer, 26

C++, 6
C-task, *see complex task*
C89, 5
C90, 5
C99, 6
cache, 20, 40, 134
    analysis, 133
        abstract interpretation, 134
        implicit path enumeration, 134
cache conflict graphs, 134
cacheanalysis
    model checking, 134
CC, *see clock cycle*
CCG, *see cache conflict graphs*
CFG, *see control-flow graph*
clock cycle, 35
code
    representation, 9
    transformation, 9, 10
code complexity, *see complexity*
code generator optimizations, 18
coding guidelines, 31
compiler, 10
compiler optimizations, 16
complex task, 16
complexity, 13, 29
concurrency, 14
conditional expression operator, 8, 64, 109, 113
control hazard, *see pipeline hazard*, 25, 26
control-flow graph, 12
    generation, 37
    segmentation, 52
cycle-level symbolic execution, 41
cyclomatic complexity, 30

data flow analysis, 133

simplified, 55
data hazard, *see pipeline hazard*
data hazards, 40
data types, 6
data-flow optimizations, 17
decision tree, 76
DO-178B, 31
DOM, *see dominator*
dominator, 13
dtree, *see decision tree*
dynamic memory, 7, 8
dynamic random access memory, 20
dynamic resource allocation, 28
dynamic timing analysis, 41

ECU, *see electronic control unit*
electronic control unit, 1, 14
error handling, 7
ET, *see event-triggered*, *see execution-time*
ETB, *see execution-time profile*
event-triggered, 33
execute, 23
execution-time, 35
analysis, 36
analysis framework, 37
best observed, 36
best-case, 35
hardware simulation, 43
measurements, 43
worst observed, 36
worst-case, 35
execution-time profile, 35
expression short-circuiting, 8, 64, 109, 113

feasibility, 52
floating point operations, 8, 27
function
black-boxing, 112
inlining, 106, 112
function calls, 55
functions, 9

hardware, 19
hardware optimization techniques, 19
hazard, *see pipeline hazard*
hybrid WCET analysis, 49
basic concept, 51
hyperthreading, 24

implicit path enumeration
cache analysis, 134
infeasible paths, 133
inlining, 55
instruction decode, 22
instruction fetch, 22
instruction latency jitter, 27
instructions per cycle, 25
interprocess communication, 14
interrupts, 15
IPC, *see interprocess communication*, *see instructions per cycle*
ISO/IEC-C, 6

JTAG, 96

lexical analysis, 10
library calls, 9
lines of code, 30
LOC, *see lines of code*
loop bound analysis, 133
loop optimizations, 17
loops, 9, 56, 101, 111
1:n, 104
general, 101
MP/VI, 56, 69, 103
nested, 101
SP/CI, 56, 67, 103
SP/VI, 56, 69, 103
specialization, 101
tree-based timing schema, 137

MAM, *see memory acceleration module*
measurement-based WCET analysis, 49
measurements
comparison of methods, 138
observed execution-times, 36
memory acceleration module, 96
memory access, 23
memory allocation, 7, 8
memory management unit, 21

MISRA, *see motor industry software reliability association*
MMU, *see memory management unit*
model checking, 54
  abstraction-based, 131
  bounded, 130
  cache analysis, 134
  symbolic, 130
motor industry software reliability association, 31

operating system, 13, 14
OR, 8, 64, 109, 113
OS, *see operating system*
other compiler optimizations, 18
out-of-order execution, 25
overestimation, 36

parse tree, 10
path, 12
path coverage, 30
PDOM, *see postdominator*
pipeline, 21–25, 40
  hazard, 24
  stall, 24
pointers, 8
postdominator, 13
predicated execution, 26
presburger analysis, 133
program segment, 13
program segments, 52
PS, *see program segment*

RAC, *see resource allocation criterion*, *see resource allocation criterion*
real-time system, 33
  event-triggered (ET), 33
  hard, 33
  soft, 33
  time-triggered (TT), 34
register write back, 23
resource allocation criterion, 29, 136
response time, 34
RT, *see response time*
RTS, *see real-time system*

S-task, *see simple task*
safe upper bound, 36
scalar, 24
segmentation, 13, 52
signals, 15
simple task, 16
simultaneus multithreading, 24
single path conversion, 129
speculative execution, 25
static analysis
  flow analysis, 38
  hardware modelling, 40
  high level, 38
  low level, 40
static random access memory, 20
static WCET analysis, 38
structs, 8
structural hazard, *see pipeline hazard*
superscalar, 24
symbolic execution, 133
syntactical analysis, 133
syntax analysis, 37
system calls, 14

target platform, 13
test results, 111
time-triggered, 34
timing anomalies, 28, 135
timing relevant dynamic computer state, 137
timing relevant system state, 137
TRDCS, *see timing relevant dynamic computer state*
TRSS, *see timing relevant system state*
TT, *see time-triggered*
type checking, 7

unions, 8

WCET, *see worst-case execution-time*
WCET analysis
  hybrid, 49
  measurement-based, 49
  static, 38
WOET, *see worst observed execution-time*

worst observed execution-time, 36
worst-case execution time
  overview of analysis tools, 126
worst-case execution-time, 35
  calculation, 46
    implicit path enumeration technique (IPET), 48
    path-based, 48
    syntax-based, 46
    tree-based, 46
  hybrid analysis, 121
  static analysis, 116
write back, *see register write back*

Appendix A

List of Abbrevations

ACET .... average-case execution-time, 35
AI .... abstract interpretation, 39
ALU .... arithmetic logic unit, 22
ARINC .... avionics application software standard interface, 31
AST .... abstract syntax tree, 10
BB .... basic block, 11
BCET .... best-case execution-time, 35
BOET .... best observed execution-time, 36
C-task .... complex task, 16
CC .... clock cycle, 35
CCG .... cache conflict graphs, 134
CFG .... control-flow graph, 12
cfg .... cfg data structure, represents a single CFG node, 62
DOM .... dominator, 13
DRAM .... dynamic random access memory, 20
dtree .... decision tree, 76
ECU .... electronic control unit, 14
ET .... event-triggered, 33
ET .... execution-time, 35
ETB .... execution-time profile, 35
GiB .... Gibibyte, 10^{30} Bytes, 20
IPC .... instructions per cycle, 25
IPET .... implicit path enumeration technique, 46
kB .... kilobyte, 1.000 Bytes, 20
KiB .... Kibibyte, 10^{10} Bytes, 20
LOC .... lines of code, 30
MAM .... memory acceleration module, 96
MB .... Megabyte, 1.000.000 Bytes, 20
MiB .... Mebibyte, 10^{20} Bytes, 20
MISRA .... motor industry software reliability association, 31
MMU .... memory management unit, 21
NOP .... No Operation, delay instruction, 45
OS .... operating system, 14
PDOM .... postdominator, 13

PS .... program segment, 13
RAC .... resource allocation criterion, 29
RAC .... resource allocation criterion, 136
RT .... response time, 34
RTS .... real-time system, 33
S-task .... simple task, 16
SRAM .... static random access memory, 20
stn .... Syntax Tree Node, 62
TRDCS .... timing relevant dynamic computer state, 137
TRSS .... timing relevant system state, 137
TT .... time-triggered, 34
WCET .... worst-case execution-time, 35
WOET .... worst observed execution-time, 36
xml .... Extended Meta Language, 78

Appendix B

Acknowledgements

Most of the implementation work as well as the writing of this thesis has been done using free software from the GNU project running under the Linux kernel. This work would not have been possible without the help of these great applications.

“Nanos gigantum humeris insidentes” and as scientists are dwarfs standing on the shoulders of giants so are software developers worldwide using and implementing free software.

Special thanks go to the following projects which provided the development library as well as a valuabe code base on which to build a project.

- The GNU project for their complete C toolchain including *gcc*, *cpp*, *as*, *ld*, *binutils*, *make*, the programming libraries *libc*, *libstdc++*, the compiler tools *flex* and *yacc* and many more invaluable tools.
- The *Linux Kernel* Project for providing the development OS.
- The *CVS* development team.
- *Xemacs* and *KDevelop* for providing the programming environment.
- *Kile* and LaTeXfor the superb type-setting environment.
- *GraphViz* for the graph plotting application *dot*.
- *PERL* for providing the scripting environment.
- The *CBMC* model checker.
- The *GLPK* (GNU Linear Programming Kit)
- The *Cxxtools* C++ tools library
- The *PCRE* Perl Compatible Regular Expressions Library

Bernhard Rieder

Curriculum Vitae

Personal Information

Nationality:	austrian
Date of Birth:	03.11.1972
Place of Birth:	Schwarzach im Pongau
Marital Status:	unmarried
Address:	Forstgasse 16, A-5500 Bischofshofen
Reachability:	e-mail: bernhard@ratte.dhs.org
Military Service:	completed

Education

6/'03-ongoing	Vienna University of Technology, Ph.D. Thesis, "Measurement-Based Execution Time Analysis of ANSI-C Applications"
10/'03-ongoing	Medical University of Vienna, Human Medicine (2. section)
10/'93-6/'03	Vienna University of Technology, Computer Engineering, Master Thesis: "*Xerxes Error Behavior*", A simulation and theoretical analysis of the error behaviour of the Xerxes encoding using systematic fault injection
9/'86-6/'92	Technical high school Braunau, Graduated in Electrical and Power Engineering with distinction
9/'82-6/'86	Secondary school, Privatgymnasium St. Rupert, Bischofshofen

Professional and Practical Experience

11/'06-11/'08	Vienna University of Technology (VUT), project assistant on the ATDGEN project, execution-time analysis of C-applications with function calls and loops and implementation of an execution-time analysis framework for the ARM9 target platform
4/'04-8/'05	VUT, project assistant on the MoDECS project, worst-case execution-time analysis of real-time C-applications, implementing of an execution-time analysis framework for embedded real-time applications in C/C++ for the HCS12 target platform
6/'01-2/'04	Decomsys GmbH, Vienna, software- and hardware-design using Perl and VHDL on Altera FPGAs, implementation of a test framework for quality and regression tests for FlexRay communication controllers
2/'99-11/'99	Festo GmbH, Vienna, database engineer using Oracle 8 and MS SQL Server, implementing alarm- and notification modules for a database based process control application
6/'93-9/'93	R&E Weinberger GmbH, Tenneck, database engineer using Oracle 7, generating quality management and controlling reports

Voluntary Services

2/'06-ongoing	Verein Wiener Sozialprojekte (VWS) and Clinical Institute for Medical Chemistry and Laboratory Diagnostics of the Vienna General Hospital (KIMCL), Database design and data analysis for the project "Spritzen-Check III"
3/'02-6/'02	VWS and KIMCL, Database design and data analysis for the project "Spritzen-Check II"
1/'99-1/'00	VWS and KIMCL, Database design and data analysis for the project "Spritzen-Check I"

Skills

Operating Systems:	Linux, Windows
Database Systems:	Oracle, PostgreSQL, MS SQL Server
Programming Languages:	C, C++ mit STL, Java basic knowledge, Perl, Tcl/Tk, bash, awk, sed
Hardware-design:	VHDL with ModelSim, Symplify Pro and Quartus
Microcontroller:	8051-Family, AVR-Family
Bus Systems:	LIN, FlexRay, TTP
Miscellaneous:	GNU Toolchain, CVS, HTML, XML

Language Skills:

German	native
English	fluent

Publications

2008

Bernhard Rieder, Peter Puschner, and Ingomar Wenzel. Using Model Checking to derive Loop bounds of general Loops within ANSI-C applications for measurement-based WCET analysis. In *Proceedings of the Sixth Workshop on Intelligent Solutions in Embedded Systems (WISES'08)*, 2008

Bernhard Rieder and Peter Puschner. Using Hybrid Timing Analysis for ANSI-C Applications with Loops and Function Calls. In Hans K. Kaiser and Raimund Kirner, editors, *Proceedings of the Junior Scientists Conference 2008*, page 101 ff. TU Wien, 2008

Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Timing Analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, Oct. 2008

2007

Bernhard Rieder, Ingomar Wenzel, Klaus Steinhammer, and Peter Puschner. Using a Runtime Measurement Device with Measurement-Based WCET Analysis. In *Proceedings of the International Embedded Systems Symposium 2007*, pages 15–26, Amsterdam, Netherlands, June 2007

Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Cross-Platform Verification Framework for Embedded Systems. In *SEUS '07: Proceedings of the Fifth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, May. 2007

2006 Raimund Kirner, Peter Puschner, Ingomar Wenzel, and Bernhard Rieder. Portable Data Exchange for Remote-Testing Frameworks. In *accepted for the 9th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Gyeongju, Korea, Apr. 2006

2005 Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. UBooks Verlag, Dec. 2005

Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Worst-Case Execution Time Analysis. In *SEUS '05: Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Washington, DC, USA, 2005. IEEE Computer Society

Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–611, Washington, DC, USA, 2005. IEEE Computer Society

Ingomar Wenzel, Raimund Kirner, Martin Schlager, Bernhard Rieder, and Bernhard Huber. Impact of Dependable Software Development Guidelines on Timing Analysis. In *Proceedings on the International Conference on "Computer as a Tool"*, Nov. 2005

Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of Timing Anomalies in Superscalar Processors. In *Proceedings of the Fifth International Conference on Quality Software (IEEE), Melbourne, Australia*, Sep. 2005

Bernhard Rieder • Forstgasse 16, A-5500 Bischofshofen • e-mail: bernhard@ratte.dhs.org