FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Precise Measurement-Based Worst-Case Execution Time Estimation

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

by

Michael Zolda

to the Faculty of Informatics
at Vienna University of Technology

Advisor:: Priv.-Doz. Dipl.-Ing. Dr. techn. Raimund Kirner

This dissertation has been reviewed by:

(Priv.-Doz. Dipl.-Ing.
Dr. Raimund Kirner)

(ao.Univ.-Prof. Dipl.-Ing.
Dr. techn. Peter Puschner)

Vienna, September 2012

(Michael Zolda)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Michael Zolda
Rögergasse 22 / TOP 2, A-1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Michael Zolda)

Abstract

A *real-time computer system* is a computer system in which the correctness of the system behavior depends not only on the logic results of the computations, but also on the physical instant at which these results are produced. Today, most real-time computer systems are implemented as collections of software tasks that are executed concurrently on a suitable hardware platform consisting of one or multiple microprocessors or microprocessor cores. Obtaining the worst-case execution time (WCET) of each real-time task is an essential step in ensuring the correctness of such a system.

Measurement-based timing analysis (MBTA) is an easily retargetable analysis approach for estimating the WCET of a given task running on a particular target platform. In this approach, the execution times of individual task fragments are measured while the task is being executed on the intended target hardware. The observed local execution times of the individual task fragments are subsequently combined into a global WCET estimate for the task as a whole.

Industry demands WCET estimates that are close to the actual WCET. To fulfill this requirement, we must limit two opposed influences on the closeness of WCET estimates obtained by MBTA: (1) conservative WCET estimate calculation, otherwise known as pessimism, and (2) incomplete measurement coverage, also known as optimism.

Preliminary versions of individual research results found in this thesis have been published in various scientific formats. In this thesis I strengthen the following original contributions:

Context-sensitive IPET: I discuss context-sensitive IPET as a strategy for reducing the pessimism found in standard IPET.

Integration of methods: I discuss how the methods for reducing optimism and pessimism that have been developed within the FORTAS project have been tied together to form a complete MBTA approach that can provide closer WCET estimates than possible with standard MBTA approaches.

Experimental evaluation: I present an experimental evaluation of the FORTAS approach. The results indicate the effectiveness of the approach. Most importantly, the evaluation includes a quantitative comparison between analysis results obtained by the FORTAS tool and the corresponding results of the industrial-strength static WCET analysis tool aiT.

Kurzfassung

Ein *Echtzeitcomputersystem* ist ein Computersystem, bei dem die Korrektheit des Systemverhaltens nicht nur von den logischen Berechnungsergebnissen abhängt, sondern auch vom physikalischen Zeitpunkt, zu dem die Resultate erzeugt werden. Heutzutage werden die meisten Echtzeitcomputersysteme als Ansammlungen von Software-Tasks implementiert, welche parallel auf einer passenden Hardware-Plattform mit einem oder mehreren Mikroprozessoren oder Mikroprozessorkernen ausgeführt werden. Das Ermitteln der Worst-Case Execution Time (WCET) der einzelnen Echtzeit-Tasks ist ein essentieller Schritt bei der Sicherstellung der Korrektheit eines solchen Systems.

Die Messbasierte Zeitanalyse (MBTA) ist ein einfach retargierbarer Ansatz zur Abschätzung der WCET eines gegebenen Tasks, der auf einer bestimmten Hardware-Plattform ausgeführt wird. Bei diesem Ansatz werden die Ausführungszeiten einzelner Task-Fragmente gemessen, während der Task auf der vorgesehenen Target-Hardware ausgeführt wird. Die beobachteten lokalen Ausführungszeiten werden anschließend zu einer globalen WCET-Abschätzung für den Gesamt-Task zusammengesetzt.

Die Industrie fordert WCET-Abschätzungen, die nahe an der tatsächlichen WCET liegen. Um diese Forderung zu erfüllen, müssen wir zwei gegensätzliche Einflüsse auf die Nähe der mit MBTA ermittelten WCET-Abschätzungen beschränken: (1) die konservative Berechnung von WCET-Abschätzungen, bekannt als Pessimismus, und (2) die unvollständige Messabdeckung, bekannt als Optimismus.

Vorläufige Versionen einzelner Forschungsergebnisse in dieser Dissertation wurden bereits im Rahmen unterschiedlicher wissenschaftlicher Formate publiziert. In dieser Dissertation erweitere ich die folgenden originären Beiträge:

Kontextsensitives IPET: Ich diskutieren kontextsensitives IPET als Strategie zur Verringerung des Pessimismus in Standard-IPET.

Integration von Methoden: Ich diskutiere, wie die Methoden zur Verringerung von Optimismus und Pessimismus, die in Rahmen des FORTAS-Projekts entwickelt wurden, zu einem vollständigen MBTA-Ansatz zusammengefügt wurden, der nähere WCET-Abschätzungen liefern kann, als dies mit Standard-IPET-Ansätzen möglich ist.

Experimentelle Evaluierung: Ich stellen eine experimentelle Evaluierung des FORTAS-Ansatzes vor. Die Evaluierung umfasst insbesondere einen quantitativen Vergleich zwischen den Analyseergebnissen des FORTAS-Werkzeugs und der entsprechenden Analyseergebnisse des statischen WCET Analyseswerkzeugs aiT.

Acknowledgment

The research on which this thesis is based has received funding from the IST FP-7 research project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)" under contract no IST-2010-248828 and from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13.

To the people who joined me on various parts of my journey.

Contents

A pessimist sees the difficulty in every opportunity; an optimist sees the opportunity in every difficulty. —*Winston Churchill*

CHAPTER 1

Thesis Overview

In this introductory chapter, we provide an overview of the topic and the contributions of this thesis.

We consider the need for close estimates of the worst-case execution time of software tasks during the construction of a *real-time computer system*. We contrast the traditional methods used for worst-case execution time estimation to the relatively new measurement-based approach. We present multiple use-cases for the measurement-based approach, where it seems favorable over traditional methods. We mention the problematic effects of *optimism* and *pessimism*, which are associated with the measurement-based approach. We then point out the contributions of this thesis and close the chapter by giving an outline of the rest of the text.

1.1 What is the WCET?

The *worst-case execution time* (WCET) of a given program on a particular execution platform is the maximal amount of time the program may require to execute, from the moment it is started to the moment it completes its execution.

1.2 Why do we need WCET Estimates?

A *real-time computer system* is a computer system in which the correctness of the system behavior depends not only on the logic results of the computations, but also on the physical instant at which these results are produced [Kop97].

Today, most real-time computer systems are implemented as collections of software tasks that are executed concurrently on a suitable hardware platform consisting of one or multiple microprocessors or microprocessor cores. Obtaining the WCET of all real-time tasks is an essential step in ensuring the correctness of such a system.

In the field of WCET analysis, we consider the problem of obtaining the WCET of a given task running on a particular hardware platform [WEE+08]. It turns out that obtaining the exact WCET is infeasible in most practically relevant cases, due to the large number of possible behaviors that even a moderately complex piece of software can

exhibit on modern computer hardware. As a consequence, a major research focus of the WCET community is the development of methods for inferring WCET *estimates*.

For practical use, the WCET estimates must be *close*, i.e., they should be as close to the actual WCET as practically feasible.

The WCET of a given task is always specific to its particular executable machine code, as well as to the target hardware on which that code is executed. There is no general method that could be used to obtain the WCET, or even a reasonably close estimate thereof, of arbitrary code to be run on an arbitrary microprocessor.

1.3 Traditional Methods for WCET Estimation

The most straightforward way to obtain an estimate for the WCET of a given piece of code on a particular target platform is to observe the end-to-end execution times while that piece of code is executed under different execution scenarios, to pick the maximal observed execution time, and to multiply that value with a safety factor that is specific to the intended application. This is what is partly done in industry in the case of low-criticality systems under the term *dynamic timing analysis* [WEE+08]. However, the same approach cannot be used for safety-critical systems.

Unlike in other fields of engineering, like, for example, construction, where systems usually demonstrate mostly continuous behavior that is suitable for such forms of extrapolation, it turns out that digital systems typically demonstrate many behavioral discontinuities. As a result, the actual WCET cannot easily be extrapolated from a set of observed execution times. Situation can and do occur, where the WCET is only triggered in rare situations that cannot be adequately modeled by continuous mathematical models. For critical applications, however, such rare situations must be considered, to ensure correct, dependable, and safe operation.

The traditional approach for WCET analysis is *static timing analysis*, where a WCET estimate is calculated directly from the executable machine code. Analysis methods that follow this approach depend on a correct and precise behavioral model of the target hardware on which the code is intended to be executed. As a result, such methods are highly dependent on the intended target hardware.

Developing a static WCET analysis for a modern microprocessor is an expensive and error-prone undertaking, as such hardware usually contains an abundance of performance optimization features, like caches, buffers, processor pipelines, branch predictors, etc. Due to the intricate interaction of these features, static analysis methods are forced to make simplifying assumptions during WCET inference, which can lead to considerably imprecise WCET estimates.

1.4 Measurement-based Timing Analysis (MBTA)

Measurement-based timing analysis (MBTA) methods [BCP02, BCP03, KWRP05, WKRP09, SM10] try to alleviate the portability issue of static timing analysis. MBTA is an easily retargetable analysis approach for estimating the WCET of a given task running on a particular target platform. In MBTA,

1. the execution times of individual task fragments are measured while the task is being executed on the target hardware, and

2. the observed local execution times of the individual task fragments are subsequently combined into a global WCET estimate for the task as a whole.

A major strength of MBTA is its low retargeting effort. Retargeting requires the implementation of a target-specific driver that compiles and links the code of the task, uploads the executable machine code to the target platform, starts the execution of the code, and records the execution time of individual program parts.

Because the space of possible input data is usually enormous, it is not feasible to perform individual measurements on all possible input configurations. A subset of possible input data must be chosen. As a result, MBTA methods face the challenge of *optimism*: A WCET estimate obtained using MBTA may be lower than the actual WCET. This is very different from the situation in static analysis, which is always conservative in its assumptions about worst-case behavior, even at the cost of a potentially high overestimation of the actual WCET. A WCET estimate obtained from a correct static WCET analysis is a guaranteed upper bound of the actual WCET.

Consequently, the subset of input data on which measurements are carried out must be selected carefully. Generating suitable input data, in order to limit the extent of underestimation, is therefore a crucial issue in MBTA.

1.5 Use Cases for MBTA

The properties mentioned in the previous section make MBTA suitable for the following scenarios:

Development of soft real-time systems: A soft real-time computer system is being developed, where an occasional miss of a task completion deadline can be tolerated. MBTA is particularly expedient for systems with high resource utilization, where close WCET estimates are particularly important.

Development of mixed-criticality systems: A mixed-criticality computer system [BLS10] is being developed. For such a system, it is possible to use a mix of static timing analysis and MBTA. In such a mixed analysis approach, static timing analysis can be used to obtain WCET estimates associated with a high criticality level, whereas MBTA can be used to obtain WCET estimates associated with a low criticality level. Dynamic mixed-criticality scheduling algorithms [BBD+10, BLS10, BBD11] can then be used to guarantee that all highly critical task are timely scheduled.

Testing and evaluation of static WCET tools: A new static WCET analysis is being developed in-house, for example, because there is a need to analyze a hard real-time system, but no suitable analysis tool is available for the chosen target hardware. MBTA can then be used to test and evaluate the newly developed static analysis tool.

Cross-checking of analysis results: A static WCET analysis tool from a third party vendor is used to analyze a highly critical hard real-time system. MBTA can be used to cross-check the obtained WCET estimates.

Development of gracefully-degrading real-time systems: A hard real-time computer system is being developed, but the costs for developing a static WCET analysis is prohibitive. MBTA can then be used to obtain WCET estimates. These estimates can be used for scheduling, under the condition that the system provides a suitable safe fall-back operation mode, which is guaranteed to be entered automatically in the unlikely case of a timing error.

Design space exploration: A hard or soft real-time computer system is being developed, but there are still open design decisions. At this stage of development, MBTA can be used for design space exploration, to help the system designer in evaluating different choices of target hardware, algorithms, task factorization, etc.

1.6 Optimism and Pessimism

In MBTA, the program code under analysis is executed on the target hardware, and the execution times for different program parts are observed. In the simplest case, the considered parts are individual instructions or non-overlapping code blocks [BCP02, BCP03, WKRP09]. Another possibility is to consider individual instruction sequences occurring in the code [BCP02, BCP03]. Typically, the *maximal observed execution times* (MOET) of each program part is then taken as approximation of the block's WCET. A global WCET estimate for the whole task can then be computed by combining the local MOETs.

A crucial issue in MBTA is closeness. The quality of the WCET estimate is determined by the ability to control two diametrically opposed sources of deviations:

Pessimism: When inferring a global WCET estimate, we combine the local WCET estimates of the individual program parts in a conservative fashion, i.e., we assume that the effects that lead to local WCETs of individual program parts may accumulate during program execution, even if we have not witnessed such a situation during measurement. The rationale for choosing this form of over-approximation is that the WCET is, by definition, a pessimistic parameter of a task. Unless we have collected sufficient evidence to the contrary, we assume the existence of execution scenarios where local WCETs of different program parts accumulate. On a global level, this form of abstraction can lead to a potential overestimation of the actual WCET.

Assume, for a moment, that we knew the actual WCET of each program part. In this case, any monotonic composition method for calculating a global WCET estimate introduces some error $e_{pess} \geq 0$, such that we have $\widehat{est} = wcet + e_{pess}$, where $\widehat{est}$ is the *optimism-free* global WCET estimate, and where $wcet$ is the actual global WCET.

Optimism: In MBTA, we cannot be sure that our measurements capture all possible execution times of a given program part, for the following two reasons:

- The size of the relevant state space, i.e., the space of possible initial conditions with respect to the *timing-relevant computer state* (cf. Section 3.3) is usually prohibitively large.
- We want MBTA to be portable, so we do not want to tailor the method to the specifics of any particular target hardware.

Although MBTA applies sophisticated input-data generation techniques to achieve measurements that cover most of the possible temporal behaviors of the individual task parts, it cannot guarantee that the local MOET, on which the global WCET estimate is based, hits the local WCET. Globally, this effect can lead to a potential underestimation of the actual WCET of the task as a whole.

Unlike static WCET analysis, MBTA intentionally does not consider the implementation details of the processing hardware, in order to preserve the portability of the method. Also, checking all possible execution times of a given code block exhaustively by enumeration is not feasible in practice, due to the size of the initial state space. As a consequence, taking the MOET of each block as an estimate of the block's WCET is inherently optimistic. Assuming, again, that a monotonic composition method is used to calculate the global WCET estimate, optimism introduces an additional error $e_{opt} \geq 0$, such that $est = \widehat{est} - e_{opt} = wcet + (e_{pess} - e_{opt})$, for the WCET estimate est under a combination of optimism—introduced by relying on measurements—and pessimism—introduced by relying on overapproximative estimate calculation.

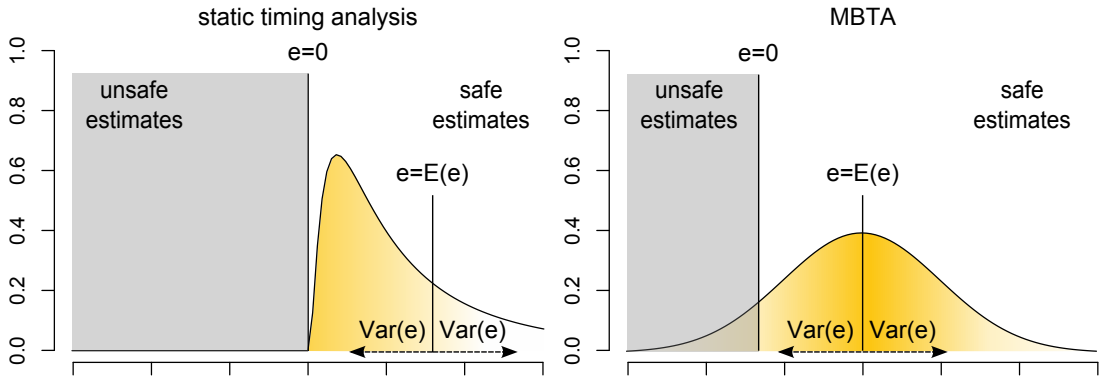

Figure 1.1: The closeness of a WCET analysis method can be quantified by reference to the variance $Var(e)$ and the estimated value $E(e)$ of the estimation error $e = est - wcet$. In the case of static analysis, the area of unsafe estimates is excluded by using overapproximation, and closeness quantifies the expected overestimation. MBTA, however, does not guarantee safety. In this case, estimates might be higher or lower than the actual WCET, and closeness quantifies the expected deviation from the actual WCET. To increase the closeness of MBTA, we must reduce both, the variance $Var(e)$ of the WCET estimate's error and the deviation $wcet - E(e)$ of the expected value of the WCET estimate's error from the actual WCET. Note that $Var(e)$ and $wcet - E(e)$ correspond to the notions of *accuracy* and *precision* in metrology, respectively.

One way to quantify the closeness of an analysis method is by reference to the variance $Var(e)$ and the expected value $E(e)$ of the estimation error $e = est - wcet$, as illustrated by Figure 1.1. To increase the closeness of MBTA, we must reduce both, the variance $Var(e)$ of the WCET estimate's error and the deviation $wcet - E(e)$ of the expected value of the WCET estimate's error from the actual WCET. This can be achieved by reducing pessimism and optimism simultaneously.

1.7 Contributions

A major requirement for a good MBTA tool that is in industrial practice is that it should be able to produce close WCET estimates, i.e., estimates that are as close to the actual WCET as practically feasible. To achieve this, the analysis must limit the influence of both, optimism and pessimism. Moreover, such a practical tool should be easy to deploy and use.

In this thesis, I condense and extend research results that have been obtained within the FORTAS project. The goal of the FORTAS project was foundational research into new methods for MBTA, which should pave the way for the development of industrial-strength MBTA tools. Central research topics that were treated within the FORTAS project are new strategies for reducing both, pessimism and optimism, and the iterative refinement of analysis results. On a higher level, the FORTAS approach aims at tying together all these methods into an integral analysis approach.

Various individual results from the FORTAS project have been published in prior scientific publications [BT08, ZBK09, ZBK10, BZTK11, BZK11, ZBK11, vHHL+11]. In this thesis I strengthen the following original contributions[1]:

Context-sensitive IPET: In Section 4, I discuss context-sensitive IPET as a strategy for reducing the pessimism found in standard IPET.

Integration of methods: In Chapter 6, I discuss how the methods for reducing optimism and pessimism that have been developed within the FORTAS project have been tied together to form a complete MBTA approach that can provide closer WCET estimates than possible with standard MBTA approaches.

Experimental results: In Chapter 7, I present an experimental evaluation of the FORTAS approach. The results indicate the effectiveness of the approach. Most importantly, the evaluation includes a quantitative comparison between analysis results obtained by the FORTAS tool and the corresponding results of the industrial-strength static WCET analysis tool aiT.

1.8 Guide to this Thesis

Chapter 2—Real-Time Computer Systems: In this chapter, we review the basic concepts of real-time computing. In Section 2.1, we start with the definition of

[1]Notice: The results presented in Chapter 5 have mostly been obtained through the research effort of Sven Bünte, as part of our research collaboration within the FORTAS project. The author of this thesis provides this overview for the sake of completeness, but does not claim any scientific results in that chapter as his own.

a real-time system. In Section 2.2, we discuss available design options for real-time computer systems. In Section 2.3, we take a look at the development process of critical systems. In Section 2.4, we review the concept of a task. We then review task scheduling on uniprocessors, in Section 2.5, and on multiprocessors, in Section 2.6.

Chapter 3—Introduction to WCET Analysis: This chapter presents the preliminaries of WCET analysis. After considering several simplifying restrictions on the hardware and software under analysis that are usually assumed for WCET analysis (cf. Section 3.1), we discuss the influence of modern high-performance microarchitectures on WCET analysis in Section 3.2. In Section 3.3, we consider what parts make up a computer's state, and how these parts may affect the temporal behavior of a running program. We then consider the software development process in Section 3.4, and its implication on WCET analysis in Section 3.5. The traditional approach for WCET analysis follows the idea of splitting the program under analysis into smaller parts and determining local WCET estimates for these individual parts. In Section 3.6 we discuss different splitting choices. Processor-behavior analysis, which we discuss in Section 3.7, is concerned with determining WCET estimates for the individual program parts. Control-Flow Analysis, which is discussed in Section 3.8, is concerned with determining the feasible sequences in which the individual program parts can be executed. Estimate calculation, which is concerned with combining the local WCET estimates obtained by processor-behavior analysis into a global WCET estimate for the whole program, is discussed in Section 3.9. We then discuss concrete methods for WCET estimate calculation, in Sections 3.10, 3.11, and 3.12. In Section 3.13, we discuss the general principles of MBTA. In Section 3.14, we discuss the MBTA issue of generating suitable input data. We conclude the chapter with a discussion of measurement, in Section 3.15.

Chapter 4—Reducing Pessimism: In this chapter, we develop a method for reducing pessimism in MBTA. In Section 4.1, we consider how pessimism arises due to overapproximation of the systems behavior. In Section 4.2, we consider a particular property found in many widely used estimate calculation methods that we need later to ensure our method for reducing pessimism can never increase pessimism. In Section 4.3, we consider that the state of a system can be viewed as a summary of its execution history, which allows us to obtain partial state information from execution histories. In Section 4.4, we present context-sensitive IPET, a generic estimate calculation method. In Sections 4.5 through 4.9, we present a concrete instantiation of context-sensitive IPET that is suitable for practical applications.

Chapter 5—Reducing Optimism: In this Chapter, we describe a method for reducing optimism in MBTA. In Section 5.1, we identify measurements as source of optimism in MBTA. In Section 5.2, we present FROO, an input-data generation technique that seeks to reduce the optimism of MBTA through an increase of the measurement coverage of temporal behavior. In Section 5.3, we present an evaluation for FROO. In Section 5.4, we describe some limitations of FROO.

Chapter 6—FORTAS: In this chapter, we present the FORTAS approach for high-precision MBTA. In Section 6.2, we review the features that distinguish the FOR-

TAS approach from standard MBTA methods. One particularly innovative feature is iterative refinement, which we review in Section 6.3. In Section 6.4, we review the modular software architecture of the FORTAS tool.

Chapter 7—Experiments: In this chapter, we present an experimental evaluation of the FORTAS approach. We first describe the target platform that we used in our evaluation. In Section 7.1, we present the TriCore TC1796 microprocessor. In Section 7.2, we present the OCDS debugging interface, which provides the technical prerequisites for collecting timed execution traces without exerting a probe effect on the system. In Section 7.3, we present the TriBoard TC179X Evaluation Board that we used as a platform for the TC1796. In Section 7.4, we present the benchmarks that we used. In Section 7.5, we explain the design of our experiments. In Section 7.6, we present the results of our evaluation.

Chapter 8—Conclusion and Outlook: This chapter closes the thesis, with the conclusion formulated in Section 8.1, and an outlook to future research in Section 8.2.

CHAPTER 2

Real-time Computer Systems

In this chapter, we review the basic concepts of real-time computing.

A real-time system is a computer system in which the correctness of the system behavior depends not only on the logic results of the computation, but also on the physical instant at which these results are produced.

In Section 2.1, we start with the definition of a real-time system. We point out the fundamental difference between *hard* and *soft* real-time systems. We also review the related concepts of *safety-critical computer systems*, *mission-critical computer systems*, *mixed-criticality computer systems*, *cyber-physical systems*, and *hybrid systems*.

Next, we consider design and development of real-time systems. In Section 2.2, we discuss available design options for real-time computer systems. In Section 2.3, we take a look at the development process of critical systems.

We then consider multi-tasking in real-time systems. In Section 2.4, we review the concept of a task, which is the fundamental building block of a modular real-time system. We then review task scheduling on uniprocessors, in Section 2.5, and on multiprocessors, in Section 2.6. We review some classic *schedulability tests*—analytic criteria that traditionally form the basis for ensuring the correctness of a multi-tasking real-time system. These tests require knowledge of the WCET—or at least a WCET estimate—of each task. Schedulability analysis is thus a primary motivation for WCET estimation.

2.1 What is a Real-Time Computer System?

A *real-time computer system* is a computer system in which the correctness of the system behavior depends not only on the logic results of the computations, but also on the physical instant at which these results are produced [Kop97].

Real-time computer systems typically manifest themselves in the physical world as *embedded computer systems*, i.e., special-purpose computer systems based on application-specific hardware and software, often implemented as a self-contained physical device. Typical applications of embedded computer systems include mobile phones, video game consoles, media players, electronic control units in various kinds of ground, air, naval, and space vehicles, plant automation systems, and medical devices. Figure 2.1 shows some typical applications of real-time computer systems.

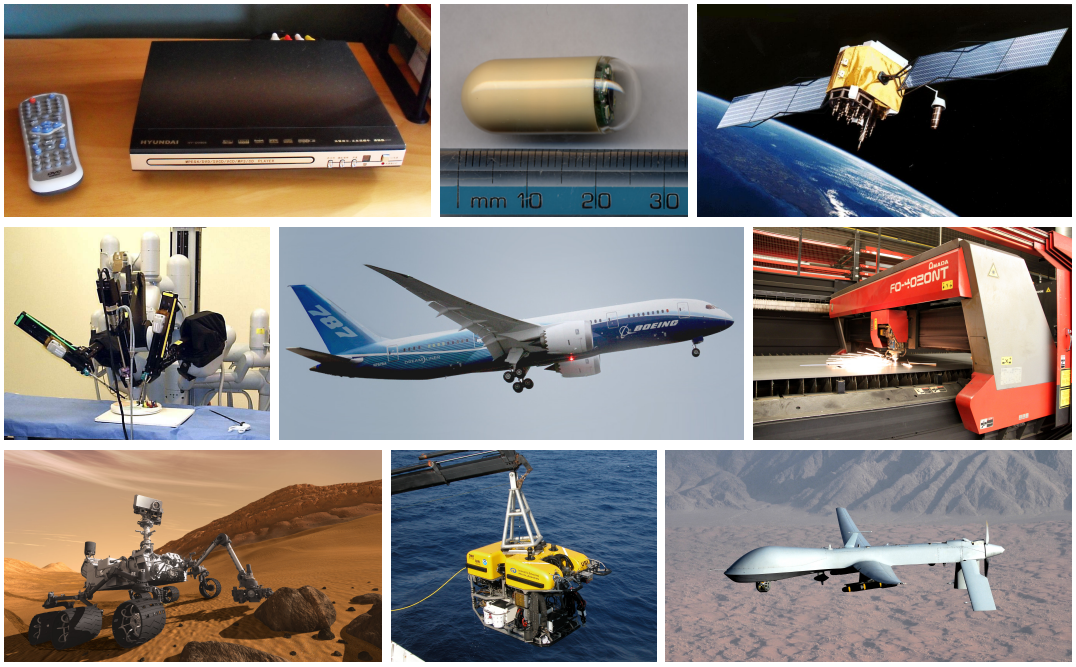

Figure 2.1: A few application examples of real-time computer systems: (a) DVD player, (b) capsule endoscope, (c) GPS satellite, (d) laproscopic surgery robot, (e) civil airplane, (f) industrial laser cutter, (g) planetary rover, (h) remotely operated underwater vehicle, (i) unmanned aerial vehicle.

Most embedded computer systems interact tightly with their physical environment. As a consequence, they are usually subject to *real-time constraints* that are imposed by the real-world objects that they interact with.

For some applications, the imposed real-time constraints are more rigid than for others. For example, an embedded controller for the combustion engine of a car must be able to inject the correct amount of fuel into the combustion chamber precisely at the right instant in time when the piston in the cylinder reaches the right position. A failure in timing can severely impact the efficiency of the engine, or even cause physical damage to the point where the safe operation of the car is compromised [Kop97]. In avionics, flight control software must execute within a fixed time interval in order to accurately control the aircraft [DB11]. Temporal constraints of this kind are called *hard real-time constraints*, and computer systems that are subject to at least one hard real-time constraint are called a *hard real-time computer systems* [Kop97].

On the other hand, there are also many embedded systems that, while still classified as real-time, are subject to less rigid timing constraints. A typical example are media players, where a voluminous stream of data needs to be processed in real time, but where an occasional miss of a temporal deadline can be tolerated, as it would merely lead to a degradation in service quality. For example, the dropping of a single frame from a video stream due to resource overload is hardly noticeable to the human viewer. However, if too many frames are dropped, the service quality of the system is impeded to a point where the system would be considered unusable. Temporal constraints of this kind are called *soft real-time constraints*, and computer systems that are subject to

at least one soft real-time constraint—but no hard real-time constraint—are called *soft real-time computer systems* [Kop97].

Occasionally, a further distinction is made between *hard real-time computer systems* and *firm real-time computer systems*, in which case a hard real-time computer system is defined as a system where a failure in the temporal domain can lead to a catastrophe, whereas in a firm real-time computer system this is never the case [Kop97].

A computer system that must operate correctly to ensure the safety of people and the environment is called a *safety-critical computer systems* [Sto96, Kni02]. Any hard real-time computer system is therefore, by definition, also a safety-critical computer system. On the other hand, a *mission-critical computer system* is a computer system where a failure can degrade or prevent the successful completion of an intended operation [Fow04]. An example of a mission-critical computer system would be an electronic trading system. In the special case of algorithmic trading [KK07b], these systems also classify as soft real-time systems.

An example of a safety-critical system is the flight control system of an unmanned aerial vehicle (UAV). A malfunction of this system may cause the vehicle to crash and cause direct damage to people and the environment. On the other hand, the navigation system might be considered less critical. A failure of this system may cause the vehicle to fail on its mission, but—in the case of a carefully designed system—it will still be possible to control the vehicle and safely abort the mission.

A *mixed-criticality computer system* is a *safety-critical computer system* that contains one or more parts that are considered non-critical, or less critical than the other parts of the system [BLS10]. Coming back to the previous example, the UAV's control system is a mixed-criticality system, if the flight control and navigation are integrated into a single system.

We use the more general term *critical systems* to refer to the union of safety critical and mission-critical systems. Since a mixed-criticality system is a special kind of safety-critical system, it follows that a mixed-criticality system is also a critical system, but, on the other hand, not every critical system needs to be a mixed-criticality system.

Rather recently, the term *cyber systems* has been coined to designate systems that perform computational tasks—in particular digital computer systems. This contrasts *physical systems*, which involve the transformation of matter and energy. Such systems subsume, for example, mechanical, chemical, or biological systems.

Accordingly, the term *cyber-physical system* designates a system that combine physical and cyber components, where the cyber components are usually embedded computer systems. Since these cyber components typically need to interact with the physical components in timely manner, a cyber-physical system is usually a real-time system. Moreover, networked communication is frequently considered as a salient ingredient of a cyber-physical system. The theory of cyber-physical systems views such systems as an integrated whole, providing tools and techniques for modeling, designing, and analyzing them.

Physical systems are characterized by their apparent *continuous dynamics*, which can be modeled by continuous mathematics. On the other hand, cyber systems—which are usually implemented using digital computer hardware—are characterized by *discrete dynamics*, which can be modeled by discrete mathematics. Systems that exhibit both, continuous and discrete dynamics, are known as *hybrid systems*.

2.2 Design Options

Most real-time computing systems are implemented on special-purpose embedded hardware that is suited to their application-specific resource requirements. There are three basic options for realizing the computational part of a real-time computing system. In practice, systems may be designed using a mix of these options:

Hardware Solution The first option is to implement the computational functionality completely in application-specific hardware. The result is an ASIC that is subsequently produced by a semiconductor manufacturer. Such application-specific hardware can be highly optimized to perform application-specific tasks effectively. However, once the ASIC has been produced, its design cannot be modified any more, which rules out any adjustments that might turn out necessary or beneficial at a later time. Moreover, the production of an ASIC is usually only cost-effective in high volumes.

Software Solution At the other end of the hardware/software spectrum, the computational functionality can be implemented in software running on an off-the-shelf embedded microprocessor. Typical microprocessor families that are found in embedded devices are the MPC5500 processor series from Freescale Semiconductors, the ARM processor series from ARM Ltd., the TriCore processor series from Infineon Technologies, the AVR processor series from Atmel Corporation, and the Cortex processor series from NXP Semiconductors, to name only a few common examples.

Reconfigurable Hardware Solution The third option is to use a FPGA. These programmable devices offer the flexibility of an ASIC in the sense that they allow for the implementation of arbitrary, user-defined logic functions. This flexibility includes the possibility of implementing complete *soft-core* processors [TAK06]. Unlike an ASIC, an FPGA can be reprogrammed, allowing for design modifications long after the system has been manufactured, for example as a part of routine system maintenance. Taking advantage of this technology, *reconfigurable instruction set processors* [BL00], for example, are able to dynamically adapt their instruction set to the application being executed. More generally, *reconfigurable computing* [Bob07] embraces the idea of routinely changing the hardware design at runtime. A computer system that implements the idea of reconfigurable computing can, for example, dynamically add special-purpose processing components as they are needed.

Identifying a balanced design that combines the benefits of these three basic solutions is a fundamental problem that is addressed in the discipline of *hardware/software codesign*.

In this thesis, we are concerned with analyzing the execution time of individual pieces of software running on a microprocessor, i.e., we are dealing with systems that have been designed according to the second design option. Methods for execution-time analysis of hardware, which are needed in the case of a hardware solution, are, for example, presented in [BC09]. Concerning reconfigurable hardware, some preliminary ideas have been presented in [KBZ10].

2.3 Dependability and the Development Process

Many real-time systems perform critical tasks during their interaction with the physical world. Consequently, such systems are usually subject to various kinds of *dependability* requirements.

Besides dependability requirements like reliability, maintainability, and availability of the system, one of the most critical requirements is usually safety, meaning that the system must not exert damage to people or the environment. Systems like flight-control systems, traffic-signaling systems, control systems for nuclear power plants, and many medical systems are considered to be highly safety-critical.

To ensure that a critical system is safe, the developers must follow a rigid design process that includes planning, implementation, review, and reporting [BP82, Fow04]. One of the most widely used development life-cycle models for the design of critical systems is the V-model [STA89, Sto96]. This model depicts the major processes involved in the design of a critical system, as well as the primary flow of information between these processes. Figure 2.2 shows a generic version of the model, as presented in [Sto96].

The information flow can be interpreted to indicate a natural sequencing of individual development phases, although, in practice, the development does not need to be strictly linear. For example, it is possible to make use of pipelining techniques to speed up the development process.

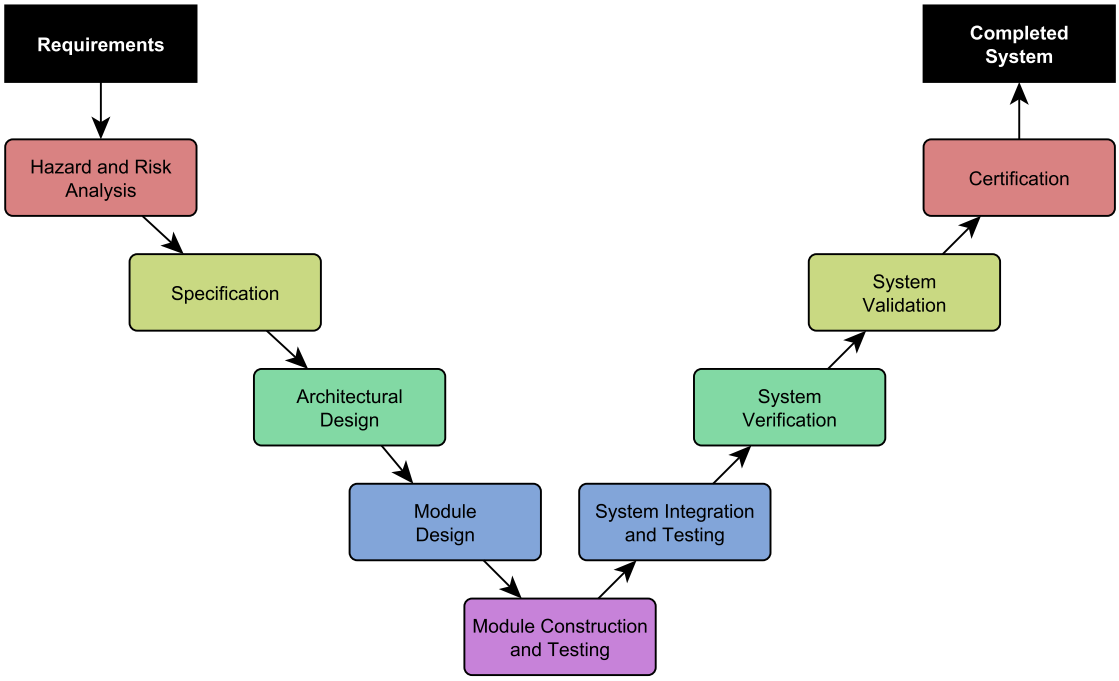

Figure 2.2: The V-model is a typical development life-cycle model that is applicable for the design of critical systems. It depicts the major processes in the development, as well as the primary flow of information between them. This particular version of the model is adopted from [Sto96].

As indicated in the model, critical systems must typically receive approval by an

official, independent certification agency. During the certification process, the manufacturer of the system is obliged to convincingly demonstrate the safety of his system. Because timing errors can affect the correct behavior of a real-time system, demonstrating temporal correctness is an important issue during certification.

Besides safety, another dependability requirement that has recently gained increasing importance in embedded and real-time computing is security. Today, more and more systems use shared processing resources or are interconnected via various kinds of networks. In the case of critical systems, it is vital to provide mechanisms and techniques that ensure the integrity of the system in the case of a malicious attack.

2.4 The Task Abstraction

Modern real-time systems are usually highly complex artifacts. To avoid unnecessary complexity in the engineering process and to keep the systems maintainable, the software of a real-time system is hardly ever designed as a monolithic piece of code. Rather, a typical design is based on a collection of cooperating *software tasks* that together serve to provide the intended service. This simplifies the isolation and correction of errors, maintenance, and the later addition of new functionality.

According to [Kop97], a task is the execution of a sequential program that starts with reading some input and state data, and terminates with some output and an update of state data.

The execution time of a given task does not include waiting times that are caused by synchronization with other tasks. As these delays depend on the progress of other tasks, they are better addressed at the scheduling level (cf. Sections 2.5 and 2.6). For execution time analysis, it is then safe to assume that all synchronization operations complete without waiting.

On systems that allow *task preemption*, a given task may be interrupted during execution, and resumed at some later time. In the meantime, other tasks or operating system software may execute on the same hardware. Such a behavior is problematic for WCET analysis, because preemption can normally occur at almost arbitrary points in a task, and any code that is executed between the preemption and subsequent resumption of a task may influence its execution time through non-functional dependencies. For example, in the presence of an instruction cache, tasks may influence each other's execution time by accessing conflicting addresses.

Depending on the application, a real-time system can be subject to different kinds of temporal constraints. Most commonly, the system is expected to deliver a certain service within a specified interval of time, i.e., at a given reference point in time t, the system must deliver its service within a *maximal response time* $RESP$. To do so, all the corresponding tasks in the system that are associated with the particular service must complete within the allowed time interval $[t, t + RESP]$.

2.5 Task Scheduling on Uniprocessors

In a multi-tasking real-time system there are multiple tasks—subject to different timing constraints—that need to be executed in a concurrent fashion. To coordinate the arising competition for processing resources, there is a need for an execution *schedule*, a plan

that determines at what times each task is to be executed on which processing unit, such that all tasks can finish execution on time. Such a schedule can be created either statically before the system is started, or dynamically during system execution:

- In *static scheduling* the temporal assignment of tasks to processing units is performed before the system is started, usually at the time when the system is designed.

- In *dynamic scheduling* a software scheduler, which executes alongside normal system tasks, decides at runtime which task should be assigned to which processing unit at any given point in time.

A static schedule is a fixed temporal assignment of tasks to processing units. This means that each task has one or more fixed time windows during which it is allowed to execute. The size of the time window(s) determines a maximal amount of time for which each task may run.

Once a WCET estimate is known for all tasks, it becomes possible to check whether a given static schedule allows each task to successively complete its purpose. If that is not the case, knowledge of the WCET estimates allows the system designer to redesign the schedule, such that all task deadlines can be met.

Concerning dynamic scheduling, there exists an abundance of different strategies for real-time systems. Here we mention just four particularly popular strategies for assigning independent tasks to a single processing unit, which are: *rate-monotonic scheduling* (RMS) [LL73], *deadline-monotonic scheduling* (DMS) [ABRW91, Aud93], *earliest deadline first scheduling* (EDF) [XP90], and *least laxity scheduling* (LL) [DM89]. All four strategies base their scheduling decisions on task priorities. At run time, whenever a task completes or a new tasks becomes ready for scheduling, a pending task with maximal priority is selected and dispatched. However, the four strategies use different metrics to assign a priority to each task.

- RMS assumes that each critical task T_i becomes ready for scheduling and must be completed periodically within a fixed time period of length p_i. The priority of each task T_i is based on its period: Tasks with shorter periods receive higher priority than tasks with longer periods. The RMS strategy can easily be extended to allow for the scheduling of additional non-critical tasks by assigning them priorities below those of any critical task.

- DMS is a generalization of RMS that allows tasks with deadlines that are placed before the end of the respective task period. This enables the modeling of communication overhead and sporadic tasks [ABRW91, Aud93].

- EDF bases the priority of each task on its upcoming deadline: Tasks with shortly upcoming deadlines receive higher priority than tasks with later deadlines.

- LL scheduling—also known as *least slack time* (LST) scheduling—establishes the priority of a task based on its slack time, i.e., the difference between its remaining time budget until its deadline and its remaining execution time. By assigning higher priorities to tasks with a lower slack time, the strategy makes sure that

pending processes are deferred as long as permissible with respect to the given deadlines.

All four strategies are *optimal* for scheduling independent, i.e., non-synchronizing, preemptable tasks on single-processor systems, in the sense that they will always find some schedule that allows all critical tasks to meet their deadlines, if such a schedule exists for the given collection of tasks.

However, testing whether such a schedule exists is usually not easy. In practice, sufficient *schedulability tests* are applied to ensure that a given set of critical tasks is schedulable under all possible circumstances.

For a given collection of n tasks $T_1, \ldots, T_n$, the inequality

$$\sum_{i=1}^{n} \frac{wcet_i}{p_i} \leq n \cdot (\sqrt[n]{2} - 1)$$

forms an utilization-based, sufficient condition for the collections schedulability under the RMS strategy [LL73]. In this inequality, $wcet_i$ denotes the worst-case execution time (WCET) of task T_i, i.e., the maximal amount of time that T_i may take to execute, if started and left running uninterruptedly. To apply this test, it is therefore necessary to first determine the WCET of each task. Finding the WCET of a task is not at all trivial, though. In fact, determining the exact WCET of a given piece of executable code running on a particular hardware platform is usually intractable. The field of worst-case execution time analysis is concerned with providing suitable approximation techniques that can be used to obtain *estimates* of the WCET.

There exists a similar schedulability test for the EDF strategy [XP90]. Again, we need to know the WCET of each task, in order to apply this test:

$$\sum_{i=1}^{n} \frac{wcet_i}{p_i} \leq 1.$$

Concerning the DMS strategy, Audsley et al. present an algorithmic schedulability test in [ABRW91]. Again, the test uses the WCET of all tasks.

Lastly, let's reconsider the LL scheduling strategy. To determine the priority of a given task, LL needs to determine that task's remaining execution time. The most practical—albeit conservative—way to do this is to dynamically track how much time each task has spent executing so far, and to take the difference to the tasks WCET as a pessimistic estimate of the task's remaining execution time. Again, we need to know to WCET of each task.

2.6 Task Scheduling on Multiprocessors and Recent Trends

The four scheduling strategies we have just seen are suitable and optimal for systems with a single execution unit. However, the advent of *multiprocessor architectures*[1] on

[1] Concerning scheduling, *multi-core* architectures are normally treated just as multiprocessor architectures. For simplicity, our use of the term multiprocessor architecture shall therefore also subsume multi-core architectures.

the embedded systems market has added a new level of complexity to scheduling: Unlike scheduling algorithms for uniprocessor architectures, algorithms for multiprocessor architectures must not only decide at what time each task should be run, but also on which processor it should be run.

Solving the latter problem, which is known as the *task allocation problem*, can be especially tricky in the case of *heterogeneous multiprocessor architectures*, where each processor might have different scheduling-relevant properties. In the case of *uniform multiprocessor architectures*, all processors are identical, but may run at different clock frequencies, leading to different possible execution times of the very same task instance. In the simplest case of a *homogeneous multiprocessor system*, all processors are identical and also run at the same clock frequency. As a result, in such systems the WCET of a given task instance is the same on all processors, which simplifies scheduling.

Scheduling algorithms for multiprocessor architectures can also be classified according to whether and when the allocation of a specific processor to a particular task is allowed to change. Whereas *partitioned* scheduling algorithms use a fixed assignment of tasks to processors, *global* scheduling algorithms may allow *task-level migration*, meaning that different task instances may be executed on different processors, or even *job-level migration*, meaning that each task instance may be dynamically moved between processors, during execution.

The choice of preemption policy—preemptive, non-preemptive, or cooperative scheduling—adds yet one more dimension to the space of different scheduling strategies for multiprocessor architectures.

A recent survey of algorithms and schedulability tests for multiprocessor architectures is given in [DB11].

Lately, scheduling theory has become an even more complex topic, through the advent of energy and thermal constraints. Energy constraints are particularly relevant for battery-powered devices, which are on a limited energy budget [SKL01]. Temperature constraints have become necessary for modern processors, where very high on-chip temperatures may occur under high processing load. To ensure the integrity of the system, it is necessary to limit the operating temperature, in order to prevent system failures that may occur due to overheating [FCWT11].

Even for the most recent task scheduling algorithms being developed, the WCET remains an essential parameter. For each critical task, the WCET must be determined, in order to guarantee that the system can always provide its critical services on time.

In the next chapter, we turn to the problem of determining the WCET of a given task on a particular hardware platform.

2.7 Chapter Summary

In this chapter, we have reviewed the basic concepts of real-time computing.

A real-time system is a computer system in which the correctness of the system behavior depends not only on the logic result of the computation, but also on the physical instant at which these results are produced. Whereas in a hard real-time system, failure to produce the result of some computation at the correct time may lead to disaster, the consequences of a similar failure in a soft real-time system are more benign. The requirements of temporal correctness are thus less rigid for soft real-time systems than

for hard real-time systems. This affects the rigor of the system development process, as well as the rigor of WCET estimation.

Today, there are multiple design options available for real-time systems. A common choice are designs that are based on a multi-tasking microprocessor. In this thesis, we focus on such designs.

The traditional way to ensure the temporal correctness of such systems is to perform a schedulability test. Such a test requires knowledge of the WCET—or at least a WCET estimate—of each task. The need to perform schedulability analysis is a primary motivation to investigate methods for WCET estimation.

CHAPTER 3

WCET Analysis

In this chapter, we consider methods for estimating the WCET of a given task running on a real-world system.

First we consider several simplifying restrictions on the hardware and software under analysis that are usually assumed for WCET analysis (cf. Section 3.1).

One major challenge that WCET analysis faces today are the complex microarchitectural features of today's high-performance microprocessors (cf. Section 3.2), which introduce a considerable jitter in the execution time of individual instructions. This jitter eventually depends on the computer state (cf. Section 3.3).

To understand the way timing analysis is implemented today, it is important to consider the software development process for real-time systems (cf. Section 3.4), which has contributed to the typical layered WCET analysis approach (cf. Section 3.5): WCET analysis is traditionally based on the idea of splitting the program under analysis into smaller parts (cf. Section 3.6), determining WCET estimates for these individual parts (cf. Section 3.7), and combining these local estimates into a global WCET estimate for the whole program (cf. Sections 3.8 and 3.9). The last combination step can be performed using different techniques, of which we discuss path-based estimate calculation (cf. Section 3.10), tree-based estimate calculation (cf. Section 3.11), and the implicit path-enumeration technique (cf. Section 3.12).

Lastly, we turn our attention to the relatively new approach of measurement-based timing analysis (MBTA) (cf. Section 3.13). The two major concerns in MBTA are the generation of suitable input data (cf. Section 3.14), and recording the execution time of individual program parts by performing measurements on the actual target hardware (cf. Section 3.15).

3.1 WCET Analysis of Real Software

In scheduling theory, a task is usually viewed as an abstract process that, when active, consumes processing time in a continuous way. In a real microprocessor-based system, however, a task is defined as a program that is intended for execution by one of the system's processors. The program consists of a collection of discrete instructions that are loaded into the system's instruction memory, from where they can be fetched and

processed by the processor. There are some options in microprocessors design that fundamentally affect WCET analysis:

Synchronicity: Most of today's microprocessors are implemented in *synchronous* logic. This means that the progress of the microprocessor's operation is controlled by a global system clock. At the beginning of each clock cycle, which is typically marked by a rising edge of the system clock signal, all the processor's latches obtain a new value that remains stable until the beginning of the subsequent clock cycle. However, the assumption of a single global clock is not valid for processors with *asynchronous* logic. A prominent example are microprocessors with *globally asynchronous locally synchronous* (GALS) logic [KGGV07]. Such a processor is driven by multiple clocks that govern individual subsystems. We are not aware of any previous work on WCET analysis of software running on asynchronous microprocessors, and we are ourselves considering synchronous microprocessor designs only. As asynchronous designs are becoming more widely used nowadays, performing WCET analysis for such microprocessors should be considered an interesting research issue.

Determinism: Most microprocessors designs are *deterministic*. Given the current state and all future inputs, there is exactly one possible future behavior that the microprocessor may exhibit. Still, *non-deterministic* microprocessors have been proposed for special applications [MMS01]. As with asynchronicity, we are not aware of any previous work on WCET analysis for non-deterministic processors, and we consider WCET analysis for deterministic designs.

Instruction set constancy: Some modern microprocessor provide multiple instruction sets that may be switched between dynamically. This is not only true for modern desktop and server processors, but also for certain processors that are specifically targeted at the embedded market. A good example is the ARM7 TDMI microprocessor, which features two instruction sets: The regular `A32` instruction set provides a full range of unrestricted 32-bit instructions, whereas the `Thumb` instruction set implements a subset of restricted variants of `A32` instructions, using compact 16-bit wide opcodes. As an intended use-case, an application that is running on a device with little memory may implement most functions using the `Thumb` instruction set, while switching dynamically to `A32` for calculation-intensive code. In the present work, we consider WCET analysis for a fixed instruction set.

Frequency constancy: In traditional microprocessor designs, the global system clock runs at a fixed frequency. However, many modern power-aware designs employ *dynamic frequency scaling* and *dynamic voltage scaling* to reduce energy consumption [SKL01]. Both techniques usually imply an adaptive frequency change of the global system clock. Only little previous work is available on WCET analysis for systems with a dynamically changing clock frequency. We consider WCET analysis for a fixed clock frequency.

The assumption of frequency constancy is important for many WCET analysis methods, because it allows the calculation of the WCET from the *worst-case execution cycles* (WCEC), the maximum number of clock cycles that a given program requires to

execute, from the moment it is started, to the moment it completes its execution. The WCET can then be easily calculated by using the equation

$$wcet = wcec \cdot f_{clk}, \tag{3.1}$$

where f_{clk} is the microprocessor's constant clock frequency.

Having considered the effect of different hardware options on WCET analysis, we must also consider certain properties of the tasks that we are going to analyze:

Preemption: When a task is preempted, its is put into *ready* state, where it does not consume any processing resources. Later, it is put back into *running* state, and execution continues. During preemption, the state of the task is preserved, so that execution can later continue with the state before the task was preempted. When we consider the WCET of a preemptable task, we refer to the maximal time for which the task is actually executing, i.e., the duration of time for which it is in its *running* state.

Synchronization: Task synchronization means that two or more executing tasks together perform some joint action. This implies that all involved tasks must be ready for the joint action at the same time. If some task becomes ready for the joint action before some of the other tasks invoked, it must wait until they are also ready. Consequently, any synchronization action involves a potential delay in the execution that depends on the execution of one or more other tasks. The execution time of a given task does not include waiting times that are caused by synchronization with other tasks. As these delays depend on the progress of other tasks, they are better addressed at the scheduling level (cf. Sections 2.5 and 2.6).

Termination: Real-time scheduling requires a bound for the processing time of each task, i.e., it requires the task's WCET. This implies that any implementation of some task must necessarily terminate on all its possible inputs, whereas non-termination is considered an error. In WCET analysis, we therefore presume the termination of any program under analysis. The development of methods and tools for checking termination is the domain of termination analysis. Recent attempts in this area include methods that are based on model checking [CGP00], and static program analysis [CPR06].

Code constancy: We assume that the code under analysis remains invariant during execution. This simplifies code analysis by excluding self-modifying code, as well as any modification of the code by the operating system.

Input: The last issue that we consider is the question when a task receives its input. There are basically two options: Firstly, we may consider tasks that dynamically receive new input while they are being processed. The second option is a task that does not receive input while it is being processed. In this case, all inputs must be encoded in the task's initial state. The task then performs a calculation, and after the program terminates the output is encoded in the termination state. In our work we consider the second option. When we talk about the input of a task, we thus mean the input that is encoded in the initial state.

3.2 Microarchitectures

Traditionally, the microprocessors employed in real-time systems used to be based on rather simple microarchitectures, where the time spent on executing a particular kind of instruction—like an addition, or a branch instruction—used to be relatively stable, or even constant. As a result, many early techniques in WCET analysis relied on the simplifying assumption of a constant local WCET for individual instructions [PK89, PS93, PS97, LM97, Pus98]. Inferring a global WCET estimate for a complete task then reduced to the problem of considering the possible execution sequences of the task's instructions, and finding the one that amounts to the highest global WCET estimate. In the case of strictly sequential execution, such an estimate can be obtained as the cost-weighted sum of instruction execution counts.

WCET analysis is traditionally biased towards pessimistic estimation, i.e., it favors estimates that are upper bounds of the actual WCET, even at the cost of a high overestimation. This is a consequence of the fact that schedulability tests are usually monotonic in the WCET estimate. For example, if we take the sufficient schedulability test for the RMS scheduling strategy [LL73]

$$\sum_{i=1}^{n} \frac{wcet_i}{p_i} \leq n \cdot (\sqrt[n]{2} - 1),$$

from Section 2.5 and replace the WCET values $wcet_i$ with corresponding estimates $\widetilde{wcet_i}$, such that

$$wcet_i \leq \widetilde{wcet_i}, \text{ for } 1 \leq i \leq n,$$

then, by the monotonicity of the inequality, we obtain a tighter condition that can only be satisfied in cases where the original condition is also satisfied. Hence, we see that using a pessimistic WCET estimate in place of the actual WCET might make the test fail for some tasks sets that might be classified as schedulable for the actual WCET, but it will never lead to misclassification of a non-schedulable task set as schedulable.

As a result of avoiding underestimation, many proposed techniques suffer from considerable overestimation, which forces system designers of highly critical systems to either over-dimension their systems, or to provide appropriate fall-back mechanisms in case of a timing error.

On today's high-performance microprocessors, individual instructions may experience a considerable jitter in execution time. The concrete execution time of a given instruction depends on the processor state. This dependence can be hard to model, because multiple microarchitectural features, like pipelines, caches, branch prediction, and parallel execution, may be involved.

To illustrate the complexity of today's modern processors, consider the following description of the FreeScale MPC8641D microprocessor, taken from [CFG+10]:

> The MPC8641D is a dual-core derivate of the MPC7448, which is a complex single-core architecture employed in the avionics industry. A single core MPC7448 consists of a [*sic*] e600 core with a complex, eight-level pipeline that allows out-of-order and speculative execution and features first- and second level caches with PLRU and random replacement. Already as a single-core, this architecture is non compositional, exhibiting both domino-effects

> in the pipeline and the caches. The MPC8641D tightly couples two such cores with a single shared bus. [...] Each access, either for the instruction fetches or any data access must pass this one shared resource. Given the non-compositionality of the two cores, any clash on the shared bus during execution could trigger a domino effect. This makes the timing behavior of the entire system very unpredictable, unless interference on the shared bus can somehow be avoided.

Kadlec and Kirner [KK07a] provide a classification of the effects the different hardware optimization features can have on WCET analysis. In particular, they identify hidden interactions between different optimization features and dynamic hardware code optimization as major challenges.

In spite of the traditional bias towards overestimation in WCET analysis, it is a primary assumption of this thesis, that underestimation is permissible in many important use-cases (cf. Section 1.5), if this can help in obtaining close WCET estimates at a lower cost.

3.3 The Computer State

In Section 3.1, we have argued that we consider strictly synchronous, deterministic microprocessors that run at a constant clock frequency. Such a processor can be understood as a finite state machine that operates in discrete steps. The timing of these steps, called *clock cycles*, is governed by a global system clock that ticks with a fixed frequency. Each fetching and processing of one of the program instructions occupies some of the processor's computational resources for a certain number of clock cycles.

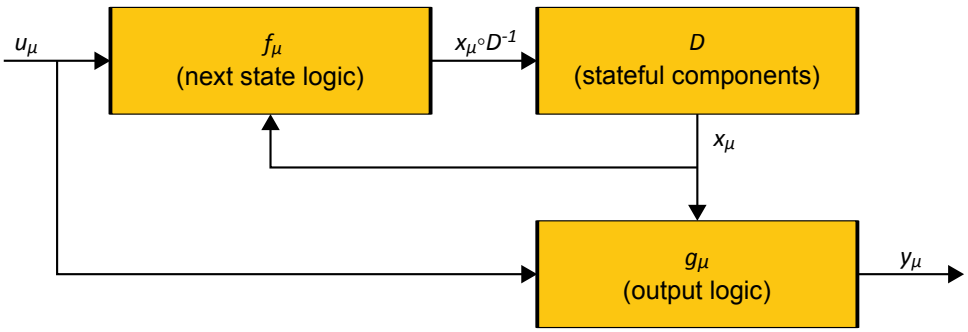

Figure 3.1: The computer as a finite state machine (Mealy variant). The *stateful components*—implemented in sequential logic—retain the current processor state. The *next-state logic*—implemented in combinational logic—computes the computer state for the next clock cycle. The *output logic*—implemented in combinational logic—computes the output.

In the formalism of signals and systems [LV01], we can describe the behavior of such

a microprocessor μ by difference equations

$$x_\mu(n+1) = f_\mu(x_\mu(n), u_\mu(n)) \quad \text{(step)}$$
$$y_\mu(n) = g_\mu(x_\mu(n), u_\mu(n)) \quad \text{(output)}$$
$$x_\mu(0) = x_{\mu 0}, \quad \text{(init)}$$

where $x_\mu : \mathbb{N} \to \mathbb{B}^l$ is a function that maps each clock cycle n to its state $x_\mu(n)$ in a state space of l bits, where $u_\mu : \mathbb{N} \to \mathbb{B}^k$ is a function that maps each clock cycle n to its input $u_\mu(n)$ in an input space of k bits, and where $y_\mu(n) : \mathbb{N} \to \mathbb{B}^m$ is a function that maps each clock cycle to its output $y_\mu(n)$ in an output space of m bits. Moreover, the equation for $x_\mu(0)$ defines the initial state of the microprocessor. Figure 3.1 illustrates how this model can be seen as the description of a finite state machine.

At the beginning of each clock cycle, which is typically marked by a rising edge of the system clock signal, all the processor's latches obtain a new value that remains stable until the beginning of the subsequent clock cycle. The total of all these latch values constitutes the *processor state*, and the Cartesian product of all latch value ranges constitutes the *processor state space*.

Among the multiple latches that make up the processor state, the *instruction address register*, also knows as *program counter* or *instruction pointer*, marks the instruction memory address of the currently fetched machine instruction[1]. Given Equation 3.1, the execution time of a piece of machine code depends on the number of clock cycles that pass, starting with the first cycle where the instruction address register obtains an address containing some instruction of that code, and ending with the last cycle before it obtains an address containing some instruction outside of the machine code under consideration.

The value that the processor latches obtain at the beginning of each clock cycle is a function of the previous processor state and the input that the processor receives via its external interfaces, defined by the deterministic, combinational processor logic. With the important exception of *reconfigurable systems* [KBZ10], the combinatorial processor logic is fixed. The number of cycles taken for executing a given piece of code therefore depends on

the initial latch state, i.e., the value of all latches just before the first instruction of the code under consideration is processed, and

the external inputs, i.e., the value of all external signals that the processor is subject to during execution of the code under consideration (we assume that the code under consideration itself has already been loaded to the instruction memory and is immutable at runtime).

For the external inputs, we can further distinguish

computer-external inputs, i.e., inputs that come from outside the computer, like, for example, external interrupts and signals received from external hardware, and

[1] In the case of a VLIW architecture, the instruction address register indicates blocks of instructions rather than individual instructions. Conceptually, operating on larger blocks of instructions is not different from operating on individual instructions.

computer-internal inputs, i.e., inputs that come from a state-bearing hardware component within the computer, like, for example, caches, memories, or other components that are connected via a processor bus.

Not all of a processor's latches are directly accessible by the machine code. Latches that are directly accessible include the various data registers, status registers, and address registers. These are relevant for the functional semantics of the code execution. We shall therefore call this part of the processor state *functional processor state*.

However, the functional semantics of the code execution can be influenced not only by the functional processor state, but also by the functional part of the state of other hardware components. For example, one such hardware component is the external data memory, which is connected to the processor via the memory-bus. The total part of the computer state that can influence the functional semantics of the code execution—which includes the functional processor state—is called *functional computer state*, or *architectural state* [KK07a].

The functional computer state may influence the execution time of the code under analysis in two ways:

Influence on control flow: The functional computer state at any given clock cycle determines the concrete sequence of instructions that is subsequently processed, which has usually a major impact on the execution time. A case where the functional computer state has no influence on the instruction sequence is *single path code* [PB02, Pus02, Pus03, Pus05], which allows, by design, only one single possible processing sequence of instructions.

Influence on individual execution times: The functional computer state at any given clock cycle may determine the number of clock cycles spent on processing subsequent instructions. A typical example is a multiplication instruction that is implemented via a shift-and-add algorithm, as found on the Motorola 6809 microprocessor. Another example is a memory-load instruction that can access multiple memory areas with different read latencies.

The rest of the computer state is transparent to the code, from a functional point of view. We therefore call this part of the computer state *hidden computer state* [KK07a]. The *hidden processor state* is the part of the hidden computer state that is also part of the processor state. This comprises latches that are not directly accessible from the code, like the latches that encode the state of the pipeline(s) and dynamic branch predictor(s). Computer components that carry part of the hidden computer state that is not also part of the hidden processor state are, for example, caches and various kinds of buffers.

Just like the functional computer state, the hidden computer state can influence the execution time of subsequent instructions at any clock cycle. For example, the state of the data caches and buffers can influence the execution time of instructions that access the data memory.

The part of the computer state that can influence the execution time of subsequently executed instruction sequences is called *timing-relevant computer state* (TRCS) [KKP09]. It may comprise parts of both, the functional and then hidden computer state.

Figure 3.2 illustrates the connection between the various parts of the computer state and gives some examples of hardware components for each parts of the computer state.

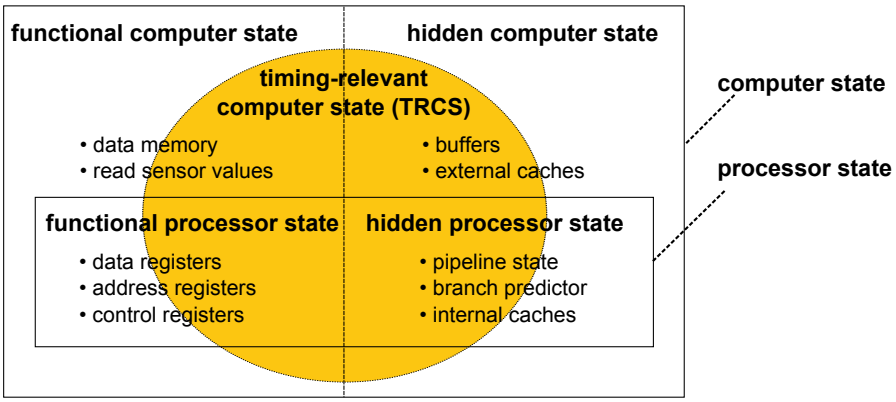

Figure 3.2: The computer state can be split into the *functional computer state*, and the *hidden computer state*, which may both contain parts that are timing-relevant. These parts form the *timing-relevant computer state* (TRCS). Functional and hidden computer state comprise functional and hidden processor state, respectively.

3.4 The Implementation of System Behavior

As already mentioned in Section 3.3, from the point of view of a hardware architect, a microprocessor operates like a finite state machine, where each transition takes one clock cycle to complete. The passage of physical time is thus an inherent aspect of the operational system model.

The stated goal of WCET analysis is to obtain an estimate of the WCET of a given task running on a particular hardware platform. To perform such an analysis, we have to take into account the properties of both, hardware and software. It is therefore important to consider how the behavior of a system is implemented:

High-level programming: Today, the software of a real-time system is usually written in a high-level programming language, like C. From the point of view of the programmer, the source code is a description of a computational process. That description ought to ultimately determine the behavior of the processor.

Compilation: Since the processor cannot execute high-level programs, a compiler is used to translate the source code to a semantically equivalent executable machine program for the target platform. To do so, the compiler must comply to both, the semantics of the high-level programming language, and the semantics of the target microprocessor's instruction set.

Execution: During the operation of the target system, the executable machine program is executed by the target processor, which implements the semantics of its instruction set via a concrete microarchitecure. By executing the machine code produced by the compiler, it, moreover, implements the behavior of the program source code.

Model-driven engineering: Model-driven engineering has rather recently introduced a fourth layer of abstraction above the high-level source code. In model-driven engineering, the system is initially described by a model that is formulated in a

domain-specific modeling language. Such a language enables engineering on a very high level of abstraction that is focused on domain knowledge, rather than on computation, as is the case for the most frequently used general-purpose programming languages. In this case, the domain-specific model, which represents the original system design, is initially translated to high-level source code that implements the behavior described by the model. Accordingly, the eventual behavior of the system is then affected by the domain-specific model.

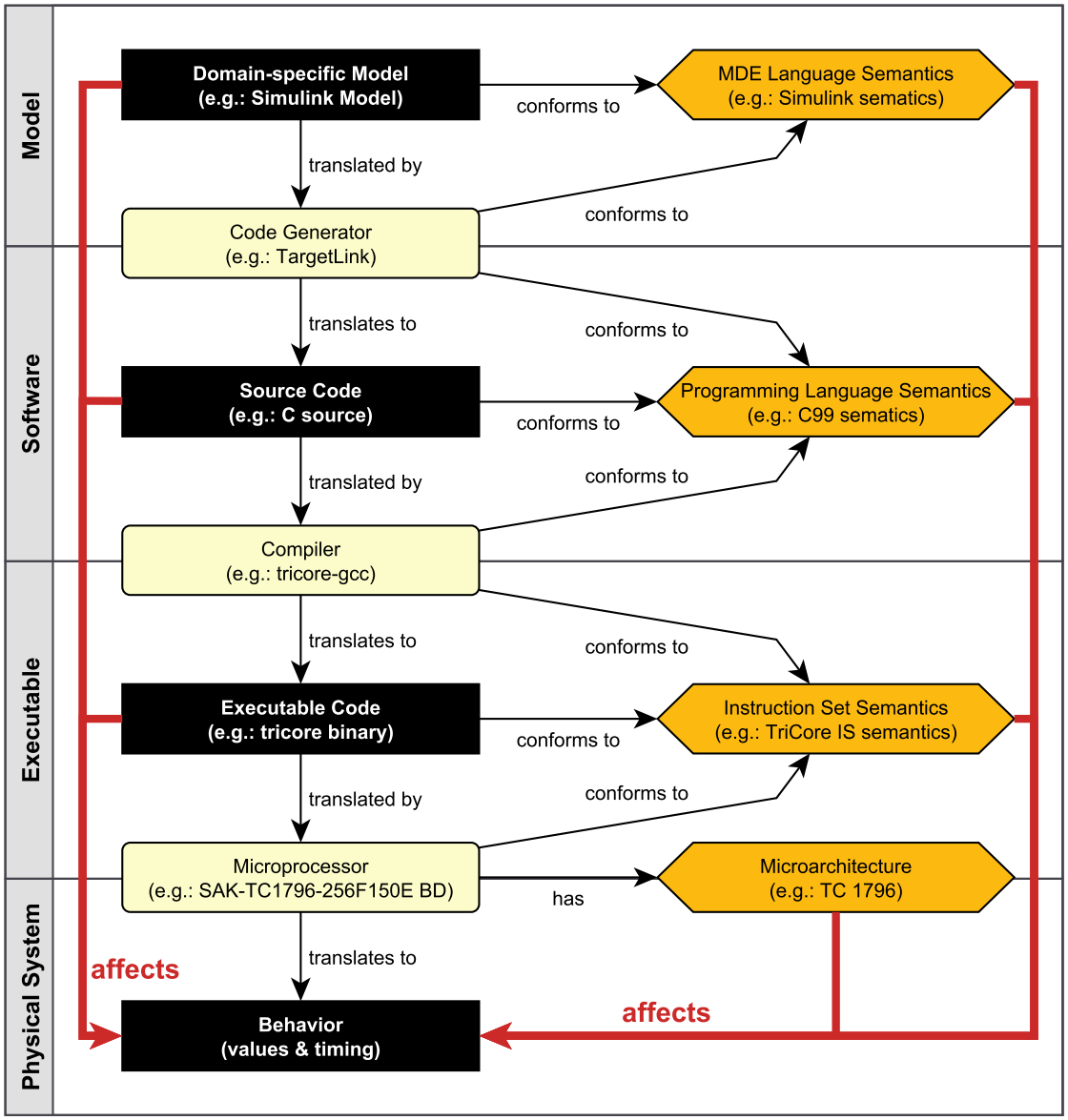

Figure 3.3: The multi-level workflow of model-driven engineering.

Figure 3.3 illustrates the multi-level workflow of model-driven engineering: Starting from a domain-specific model of the system, a code generator is used to automatically generate source code. The domain-specific model is itself based on the original system

specification (not shown), which describes the salient properties that the system under design should eventually have. The source code is subsequently fed into the target-specific compiler tool-chain. This tool-chain eventually produces machine code that can be executed on the target platform, inducing the desired behavior of the physical system, as demanded by the specification.

More importantly, the figure indicates how the eventual behavior of the system is affected by the semantics of the model and its descendants, i.e., the source code and the machine code, on the one hand, and the microprocessor's microarchitecture that implements the microprocessor's instruction set, on the other hand. The microarchitecture is determined by the microprocessor manufacturer and can normally not be changed by the system designer. It is semantically transparent in the value domain, but directly determines the temporal behavior (cf. Section 3.3).

3.5 Layered WCET Analysis

WCET analysis of software has traditionally focused on the lower three levels, i.e., the analysis approaches are designed to analyze the WCET of programs that implement the individual tasks of a real-time system.

Recently, however, there have been efforts to integrate the static timing analysis tool aiT [TSH+03] with model-driven development tools like SCADE [FHLS+08, LSHK08], ASCET [FHW+08, FHRF08], and RT-Druid, as well as with the scheduling analysis tool SymTA/S [KWH+09, FHRF08].

From the point of view of WCET analysis, the corresponding publications generally report an improvement of the closeness of the obtained WCET estimates for generated code in comparison to hand-written code. This observation is attributed to the limited set of code patterns that occur in automatically generated code.

In contrast, other works argue that WCET analysis ought to more directly exploit the information contained in high-level system models. Erpenbach and Altenbernd [EA99] have presented an approach for directly analyzing Statemate statecharts [HN96].

Kirner et. al. [KLPT00, KLFP02] have presented an approach for analyzing Matlab/Simulink models. Their approach uses a modified code generator to extract supplementary control-flow information.

Lei et. al. [JHCR09, JHRC09, JHRC10] have presented an analysis technique for Esterel [BdS91, BCE+03] programs that makes use of context information.

The FORTAS approach follows the traditional approach of targeting the program code for analysis. One benefit thereof is that the approach works independently from any particular modeling language, while retaining the possibility of coupling it with higher-level models, in a similar way as was done for aiT.

The traditional approach for WCET analysis follows the idea of splitting the program under analysis into smaller parts and determining local WCET estimates for these individual parts. Subsequently, a global WCET estimate for the complete program is inferred from the individual local WCET estimates. This also requires an analysis of how the local WCET estimates can be combined. A complete analysis approach therefore consists of the following three stages:

Processor-behavior analysis: Processor-behavior analysis (cf. Section 3.7) is concerned with the possible execution times that individual program parts, like in-

structions, instruction sequences, or code blocks may exhibit on a particular target platform, under different hardware states. The central point of processor-behavior analysis is to determine WCET estimates for the individual program parts.

Control-flow analysis: control-flow analysis (cf. Section 3.8) is concerned with determining the feasible sequences in which the individual program parts can be executed. The central point of control-flow analysis is to exclude as many provably infeasible execution sequences as practically possible from further consideration during estimate calculation.

Estimate calculation: Estimate calculation (cf. Section 3.9) is concerned with inferring a global WCET estimate of the whole program, combining the information obtained from the processor-behavior and control-flow analyses.

The rationale behind this separation is that processor-behavior analysis relies largely on the temporal semantics of the hardware, which is determined by the microprocessor's microarchitecture, whereas control-flow analysis relies largely on the functional semantics of the program code. A monolithic approach is considered prohibitively complex in most cases, and this division serves as a natural line of separation.

In general, however, processor-behavior analysis cannot always be fully separated from control-flow analysis without a loss of closeness. One particular problem is that the execution time of an individual instruction may depend on the TRCS (cf. Section 3.3), which, in turn, may be depend on which instructions have been previously executed.

The method described in Chapter 4 addresses this problem by using context information to distinguish different execution scenarios of the individual program parts.

3.6 Splitting a Program

The traditional approach for WCET analysis follows the idea of splitting the program under analysis into smaller parts and determining local WCET estimates for these individual parts. Later on, the WCET estimates of the individual parts are combined into a global WCET estimate for the whole program. The exact notion of a program part may vary from analysis to analysis. The following are key factors that affect the concrete choice:

System representation level: In Section 3.4, we have seen that a system implementation may be viewed at different representation levels. Also, in Section 3.5, we have argued that the rationale behind the separation of control-flow analysis from processor-behavior analysis is that processor-behavior analysis relies largely on the temporal semantics of the hardware, whereas control-flow analysis relies largely on the functional semantics of the program code. Control-flow analysis may be performed at different representation levels, and depending on this choice, the definition of a program part may differ: If the analysis targets machine code, program parts may, for example, be individual instructions. In the case of VLIW processing, however, it may be more useful to refer to blocks of instructions that are processed in parallel. If the analysis is performed at the source-code level, *basic blocks* [Muc97] are a natural choice for program parts. An analysis of the

domain-specific system model may refer to the model's functional building blocks. In the case of third party software, the higher level representations might not be disclosed. In that case, performing an analysis of the machine code may be the only available option.

Scale of analysis: Next, the choice of program parts also depends on the scale on which the analysis should be performed. There is a trade-off between the closeness that can be achieved by WCET estimation under given resource constraints—like analysis time—and the size of a system that can be analyzed. Basic blocks are a natural candidate for program parts in many situations. However, a more fine-grained choice—like individual statements and/or expression evaluation steps—allows for a more detailed modeling of individual timing effects, because it permits the analysis to localize them more precisely. On the other hand, to perform an analysis of huge quantities of code, it may be necessary to choose much larger program parts, like entire code fragments or even components.

Estimate calculation: Lastly, different methods for estimate calculation (cf. Section 3.9) require different kinds of program parts. Path-based estimate calculation (cf. Section 3.10) operates on instruction sequences that are allowed to overlap each other. Tree-based estimate calculation (cf. Section 3.11) requires the program parts to coincide with the leave nodes of the program's abstract syntax tree. The original version of the implicit path-enumeration technique (cf. Section 3.12) operates on contiguous code fragments that have a single point of entry, as well as a single point of exit for control flow, so-called SESE regions [JPP94]. These regions may form a partition, in which case they must cover the entire code without overlapping, or a hierarchy.

Analyzing code at the software level rather than on the executable level brings certain advantages:

Ease of flow analysis: Analyzing the control flow is usually easier for high-level code than for machine code. In particular, compiled code may contain computed jumps that are difficult to analyze. Further complications when working on compiled code may arise due to use of branch delay slots, guarded code, and VLIW, due to the embedding of data within code sections, and more [The00].

Ease of code annotation: Most analysis methods require the user to provide certain code annotations. In particular, the user must typically provide constraints—so-called *flow facts*—that describe the maximal number of iterations of loops, in cases where the analysis is unable to discover such bounds automatically. Additional flow facts can be provided to improve the closeness of the analysis results. For the user, it is usually more convenient to annotate high-level code than machine code.

The mapping between source code and machine code can, however, be compromised by optimizing code transformations that are performed by the compiler. This has two major consequences:

Mismatch of flow facts: Flow facts that are specified at the source-code level cannot be mapped directly to the machine code, but must be transformed along with

the code [Kir08, KPP10a]. This affects all analysis approaches that require the specification of flow facts, including measurement-based approaches.

Mismatch of code coverage: Test suites that meet a given structural code coverage criterion—like statement, decision, or predicate coverage—at the source-code level do not necessarily meet the same criterion at the machine-code level. Kirner has presented criteria for the preservation of structural code coverage [Kir09]. These criteria can be used to decide whether a given compiler optimization affects a given coverage criterion [KH09]. Experiments have shown that a classification of compiler optimizations can be used to restrict optimizations to unproblematic cases. Importantly, it has been demonstrated that such a restriction does not cause a significant performance penalty [KZ11].

3.7 Processor-Behavior Analysis

Processor-behavior analysis, or micro-architecture analysis, is concerned with finding the possible execution times that individual program parts, like instructions, instruction sequences, or code blocks may exhibit on a particular target platform. The central point of processor-behavior analysis is to determine WCET estimates for the individual program parts.

In the case of static WCET analysis, an upper bound of the local WCETs of individual program parts is calculated by applying techniques from static program analysis [NNH99].

A widely used technique in processor-behavior analysis is abstract interpretation. In [FMWA99], Ferdinand et al. have presented an abstract interpretation for modeling the behavior of a set-associative LRU cache. In [SF99], Schneider and Ferdinand have presented an abstract interpretation for a superscalar pipeline design. Colin and Puaut [CP00] have presented an abstract interpretation for a 2-bit saturating branch predictor.

In the case of measurement-based WCET analysis, local WCET estimates of individual program parts are obtained from measurements (cf. 3.13).

3.8 Control-Flow Analysis

Control-flow analysis is concerned with determining the feasible sequences in which the individual program parts can be executed. The central point of control-flow analysis is to exclude as many provably infeasible execution sequences as practically possible from further consideration during estimate calculation. Information about which execution sequences are feasible is called *control-flow information*, or, more concisely (but less precisely), *flow information*. Individual bits of control-flow information are sometimes called *flow facts*.

One possible source of control-flow information is the program's AST. In compiler construction, an AST is a tree structure that encodes the hierarchical syntactic structure of a program. However, the popular paradigm of *structured programming* [Dij70] demands that structure of the possible control flow within a given program should coincide with the program's syntactic structure. In such a setting the AST does not only

encode the syntactic structure, but also implicitly describes the possible control flow. Tree-based estimate calculation (cf. Section 3.11) makes direct use of this control-flow information.

A more explicit representation of control-flow information is provided by a program's CFG, which is normally obtained via static code analysis:

Definition 3.1 (Control-flow graph)**.** *A* control-flow graph G *of program* $\mathcal{P}$ *is a quadruple* $(V, E, v_{start}, v_{end})$*, consisting of a set of* nodes *or* vertices V*, a set of* edges $E \subseteq V \times V$*, a* start node $v_{start} \in V$*, and an* end node $v_{end} \in V$*, such that all other nodes are reachable from* v_{start}*, i.e.,* $(v_{start}, v) \in E^+$*, for any* $v \in V$*, where* E^+ *is the transitive closure of* E*, and* v_{end} *is reachable from all other nodes, i.e.,* $(v, v_{end}) \in E^+$*, for any* $v \in V$*. Moreover, we require that* v_{start} *is not strictly reachable from any node, i.e.,* $(v, v_{start}) \notin E^+$*, and no node is strictly reachable from* v_{end}*, i.e.,* $(v_{end}, v) \notin E^+$*, for any* $v \in V$*.*

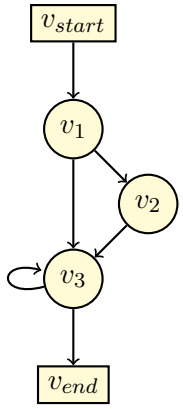

Figure 3.4: Visualization of the CFG from Example 3.1.

Example 3.1. Consider $\mathcal{G} = (V, E, v_{start}, v_{end})$ with

$$V = \{v_{start}, v_{end}, v_1, v_2, v_3\} \text{ and}$$

$$E = \{(v_{start}, v_1), (v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_3), (v_3, v_{end})\}.$$

Figure 3.4 contains an illustration of $\mathcal{G}$.

Semantically, each node in a CFG corresponds to a program part that is assumed to be executed in an atomic fashion, at suitable abstraction level. The start node v_{start} and the end node v_{end} are special, as they do not have any real program code associated with them, but are merely introduced to mark the entry and exit point of program execution, respectively. To keep things simple, we conceptually consider that these special nodes are associated with empty code that can be executed instantaneously, without exerting any side-effects. For simplicity, we will identify each node with its associated program part; for example, we might say that a node is executed, rather than saying that its associated program code is executed.

A directed edge in a CFG with a source node v and a target node w indicates that the execution of w *may* immediately follow the execution of v. On the other hand, the absence of an edge from node v to node w indicates that the execution of w *must not* immediately follow the execution of v. Consequently, the execution of any node v having exactly n immediate successors $w_1, \ldots, w_n$ *must* be followed by the execution of *exactly one* of its successors $w_1, \ldots, w_n$.

Since the start node v_{start} is not strictly reachable from any node, it may only occur at the start of an execution sequence, and since all other nodes are strictly reachable from v_{start}, it is the only possible starting point of an execution sequence. Likewise, no node is strictly reachable from the end node v_{end}, hence it may only occur at the end of an execution sequence, and since v_{end} is strictly reachable from all other nodes, it is the only possible termination point of an execution sequence.

The set of all paths that start with v_{start} and end in v_{end} overapproximates the set of possible execution sequences. Any sequence of program parts that does not form such a path in the CFG cannot occur during program execution and can thus be safely excluded from timing analysis.

The notion of a CFG that we are using is closely related to the one used in compiler construction, but there are some minor differences:

1. In compiler construction the nodes of a CFG usually correspond to maximal atomic blocks of code, so-called basic blocks. This restriction is due to performance considerations of the algorithms operating on CFGs. Since our presentation is on a conceptual level, we allow any suitable kind of program parts, as discussed in Section 3.6. It might be useful, however, to introduce similar restrictions at an implementation level.

2. Our definition of a CFG automatically excludes certain undesirable program structures, like unreachable code.

The expressive power of CFGs corresponds to regular expressions. This means that CFGs are not powerful enough to impose iteration bounds on cyclic program structures. Such bounds are traditionally provided as additional constraints over the CFG.

Many iteration constraints can be derived automatically by code analysis. Healy et al. [HSRW98] presented three different methods for bounding the number of loop iterations: one method that can determine numeric upper and lower iteration bounds for loops with multiple exits; one method for inferring symbolic iteration bounds for loops where the iteration bound depends on loop-invariant variables; and one method for handling nested loops.

Ermedahl and Gustafsson [EG97, GESL06] used a variant of abstract interpretation to determine the possible value ranges of program variables, which includes the induction variables of loops. Later Ermedahl et al. [ESG+07] presented a related approach that supports program slicing and mixed interval/congruence domains.

Holsti and Saarinen [HS02] presented an approach that tries to identify program variables that act as loop counters. By bounding both, the initial value of the loop counters and the increment per iteration, they were able to infer iteration bounds for certain kinds of loops.

Rieder et al. [RPW08] have presented a method that employs the CBMC model checker for ANSI C and C++ programs [CKL04] to find loop bounds. The do so by modifying the source code, requiring no adaption of CBMC.

Zuleger [ZGSV11] demonstrated that the size-change abstraction is an effective abstract domain for bound analysis.

If a suitable loop bound cannot be derived automatically for some cyclic program structure, then it must be supplied by an expert who has to manually inspect the source code.

Apart from determining loop bounds, control-flow analysis can also infer information about infeasible paths in acyclic code fragments [Alt96]. Approaches for control-flow analysis that are based on symbolic execution [LS98, LS99, GESL06] are, in general, capable of excluding infeasible paths, and so is the approach by Healy et al. [HSRW98, HW99].

3.9 Estimate Calculation

Estimate calculation is concerned with combining the local WCET estimates obtained by processor-behavior analysis (cf. Section 3.7) into a global WCET estimate for the whole program, according to the information obtained by control-flow analysis (cf. Section 3.8). Three basic techniques have been proposed for estimate calculation. We will briefly discuss these methods in Sections 3.10 through 3.12:

Path-based calculation: The path-based calculation method operates—to a large extent—on individual control-flow paths. Since the number of control-flow paths is usually huge, even for small CFGs, all practical path-based analysis methods really operate on path fragments, or subpaths.

Tree-based calculation: The tree-based calculation method calculates the global WCET estimate bottom up along a tree structure encoding the possible execution sequences of program instructions.

Implicit Path-Enumeration Technique: The IPET is a calculation method where the set of allowed paths is described by integer linear constraints over the execution frequency of the individual program parts. Under the assumption of a constant local WCET estimate for each program part, a global WCET estimate can be obtained by finding a maximal solution of a weighted sum of local execution frequencies.

The previous classification is based on how the program under analysis is decomposed, and how the local WCET estimates are, accordingly, combined. Another aspect that varies between different calculation methods is the value domain that is used to represent WCET estimates:

Numeric domain: In the simplest case, a WCET estimate is represented by a single, positive numeric value that encodes the duration in terms of real (“wall-clock”) time or processor cycles. Sequential execution then corresponds to numeric addition, and alternative execution correspond to taking the maximum of the corresponding set of WCET estimates.

Reservation tables: This is a more sophisticated domain, which encodes information about the occupation times of different stages in a pipelined architecture. Here, sequential execution corresponds to a concatenation operation, and alternative execution corresponds to logical disjunction [RML+94, HWH95, LBJ+95, HBL+95, KMH96, HAM+99, SA00].

Probability distributions: In the case of probabilistic WCET analysis [BCP02, BCP03], WCET estimates are encoded by probability distributions, which describe the chance that the execution time will exhibit a certain value. In this case, sequential execution corresponds to convolution. For alternative execution, Bernat et al. [BCP02] propose a maximum operator. Unfortunately, it is not clear how the resulting probability distribution should be interpreted under the latter operation. Whereas the convolution operator allows us to interpret the resulting probability distribution as describing the probability that the corresponding piece of code will exhibit a certain execution time, the same is not true for the probability distributions obtained by applying the proposed maximum operator.

3.10 Path-based Estimate Calculation

The path-based calculation method operates—to a large extent—on individual paths. Since the number of control-flow paths is usually huge, even for small CFGs, all practical path-based analysis methods really operate on path fragments, or subpaths.

One widely used strategy is to partition the CFG of the program under analysis into hammock graphs [ZD04] and to treat the resulting subpaths within each hammock on an individual basis. The partitioning itself can be performed statically, when the analysis starts [WKRP05, BC08, WKRP09, ZBK09, ZBK11], or dynamically, through a periodical merging of path information during analysis strategies that are based on abstract execution or similar techniques.

The analysis obtains a local WCET estimate for each hammock. These local WCET estimates are subsequently combined into a global WCET estimate for the whole program. This can be done using specialized longest path search algorithms [LS98, LS99, SA00, SEE01], by employing the tree-based approach [HWH95, CP00, CP01, CB02, BB06], or by using IPET [EE99, EE00, WKRP09, ZBK09, ZBK11]. Interestingly, the path-based approach can be seen as a refinement of the latter estimation methods, rather than a genuine estimation method.

3.11 Tree-based Estimate Calculation

The idea of *tree-based WCET analysis* [PK89, PS93, Pus98], which is also known as *timing schema* [Sha89, PS91], is to calculate the global WCET estimate bottom up along a tree structure encoding the possible execution sequences of program instructions.

In compiler construction, an AST is a tree structure that encodes the hierarchical syntactic structure of a program. However, the popular paradigm of *structured programming* [Dij70] demands that structure of the possible control flow within a given program should coincide with the program's syntactic structure. In such a setting the AST does not only encode the syntactic structure, but also describes the possible control flow.

In particular, the semantics of compound constructs—like sequential execution, conditional, and looping constructs—mandate that the set of allowed instruction sequences of a corresponding AST node is obtained from the instruction sequences of its child nodes via the operations of concatenation and set union.

For example, the possible execution sequences for a sub-AST corresponding to a *while*-like looping construct with an iteration bound of k could be described by the set

$$\{b \circ (s \circ b)^i \mid 0 \leq i \leq k, b \in \mathcal{B}, s \in \mathcal{S}\},$$

where $\mathcal{B}$ is the set of possible execution sequences for the evaluation of the loop condition, where $\mathcal{S}$ is the set of possible execution sequences for the loop body, where the operator $\circ$ denotes subsequent execution, and where

$$(b \circ s)^i = \begin{cases} \epsilon & \text{if } i = 0, \\ (b \circ s) \circ (b \circ s)^{i-1} & \text{if } i \geq 1, i \in \mathbb{N}. \end{cases}$$

```
int binary_search( struct DATA const * data, int x )
{ int fvalue, mid, low, up;
  fvalue=0;
  mid=0;
  low = 0;
  up = 14;
  fvalue = -1; /* all data are positive */
  while (low <= up) {
    mid = (low + up) >> 1;
    if ( data[mid].key == x ) {  /*  found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else  /* not found */
      if ( data[mid].key > x ) {
        up = mid - 1;
      }
      else   {
        low = mid + 1;
      }
  }
  return fvalue;
}
```

Figure 3.5: Source code of a binary-search algorithm in C.

As an illustration of the idea, consider the example source-code snippet in Figure 3.5, which is a C implementation of a binary-search algorithm.

If we resolve the hierarchical structure of the statements into a tree, we obtain an AST like the one depicted in Figure 3.6. Assuming that we have already obtained WCET estimates for the atomic program parts corresponding to the tree leaves (marked in black) we can easily derive a global WCET estimate for the complete program by applying a set of simple inference rules.

Each rule infers the WCET estimate for one of the possible compound constructs

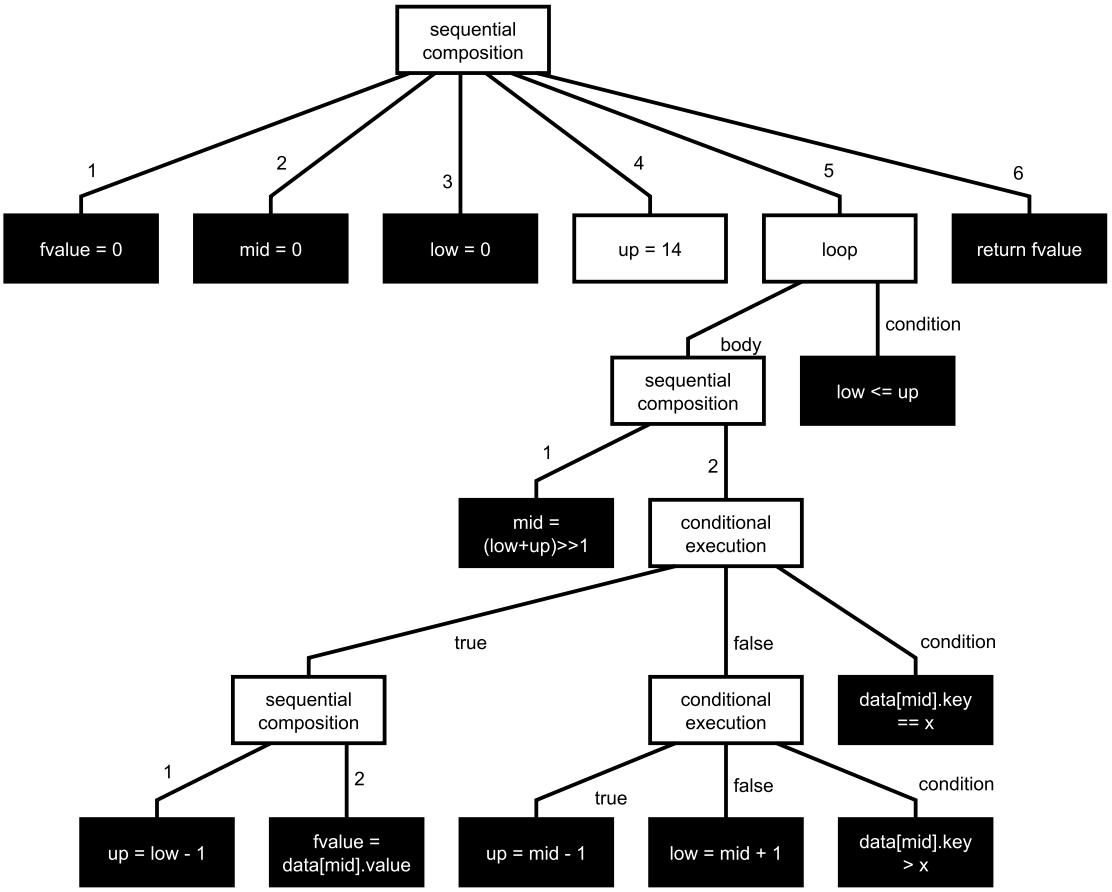

Figure 3.6: AST of the binary-search implementation from Figure 3.5. The black nodes are leaf nodes containing atomic statements, whereas the white nodes correspond to compound statements.

from the WCET estimate of the construct's child nodes. The rules mirror the restrictions imposed by the paradigm of structured programming and the semantics of the individual constructs. By applying these rules in a bottom-up fashion, we obtain a WCET estimate for the root node, which corresponds to the complete program.

The actual rules depend on the specific semantics of the used programming language and on the level of abstraction used in the modeling of the control flow. In a simple case, the inference rules for the three compound statement types of a WHILE-like language [NNH99], i.e., sequential composition, conditional execution, and while loop, could be

$$\frac{\widetilde{wcet}(S_1) = t_{S_1} \quad \widetilde{wcet}(S_2) = t_{S_2}}{\widetilde{wcet}(S_1\ ;S_2) = t_{S_1} + t_{S_2}}\ \textit{(sequential composition)},$$

$$\frac{\widetilde{wcet}(b) = t_b \quad \widetilde{wcet}(S_1) = t_{S_1} \quad \widetilde{wcet}(S_2) = t_{S_2}}{\widetilde{wcet}(\text{if } b \text{ then } S_1 \text{ else } S_2) = t_b + \max(t_{then} + t_{S_1}, t_{else} + t_{S_2})}\ \textit{(cond. exec.)}, \text{ and}$$

$$\frac{\widetilde{wcet}(b) = t_b \quad \widetilde{wcet}(S) = t_S}{\widetilde{wcet}(while\ b\ \text{do}\ S) = (k+1) \cdot t_b + k \cdot (t_S + t_{loop}) + t_{end}}\ \textit{(loop)},$$

where t_{then} and t_{else} are, respectively, the constant WCETs for transferring control to and back from the *then* and *else* branch (typically implemented by one conditional forward jump plus one unconditional forward jump), where t_{loop} is the constant WCET for transferring control to the loop body and later back to then loop test (typically implemented by one conditional forward jump plus one unconditional backwards jump), where t_{end} is the constant WCET of transferring control out of the loop (typically implemented by one conditional forward jump), and where $k > 0$ is an integer constant denoting the maximal number of iterations of the respective loop.

The pessimism of the WCET estimates that are obtained through the tree-based approach can be reduced through the use of additional flow constraints. For example, the approach presented by Puschner, Koza, and Schedl [PK89, PPVZ92, PS93, Pus98] introduces program annotations that allow the user to place constraints on the control flow within program *scopes*. Colin and Puaut [CB02] have presented a similar technique.

A scope limits the influence of a particular constraint to the control flow within a specific subtree of the AST, typically a loop. An important use case is the exclusion of infeasible paths within nested loops.

Park [Par93] proposed a special-purpose language that allows the user to concisely specify typical constraints on the occurrence of individual statements on control-flow paths. Again, based on the notion of a scope, their language provides primitives for specifying how often a particular statement should occur along a path, disallowing or enforcing the occurrence of a statement, specifying the mutual exclusion or inclusion of two statements, and more. For analysis, the user-provided constraints are translated to regular expressions, which are used calculate an overapproximation of the feasible control-flow paths. To perform the actual estimate calculation, two techniques are proposed: The first technique is based on transforming the program according to disjoint sets of feasible control-flow paths. The second technique is a variant of the tree-based approach with refined rules that consider the set of feasible paths through the current AST subtree.

Another way to extend the tree-based approach is to use a different value domain. For example, in [RML+94, LBJ+95, HBL+95, KMH96], sets of worst-case reservation tables [Kog81] have been used as domain elements, in order to model the instruction interleaving in single in-order issue pipelines, as found in RISC processors similar to the MIPS R3000. The latter work also presented a strategy for handling the state of direct-mapped caches via post-conditions for individual domain elements. In [KMH96], Kim et al. use similar post-conditions to model the state of direct-mapped data caches. Related techniques have also been used Healy, Whalley, and Harmon [HWH95, HAM+99] to model instruction interleaving and caching on a SPARC V7 processor.

Colin and Puaut [CP00] have presented a tree-based approach to take into account branch prediction, using static program analysis to classify the behavior of the branch predictor at different program locations.

The tree-based approach can also be used to calculate an estimate of the global BCET [Sha89]. It is efficient and particularly appealing due to its compositional nature, which is especially suitable in cases where a user is not interested in the global WCET estimate only, but would moreover like to investigate the estimated WCET of individual

program parts. On the downside, the approach is limited to programs that follow the paradigm of *structured programming* [Dij70], ruling out unstructured jumps.

The approach by Betts and Bernat [BB06] uses a variant of tree-based estimate calculation. However, they do not rely on the program AST, but use a special structure that is based on instruction sequences between instrumentation points.

3.12 The Implicit Path-Enumeration Technique (IPET)

The IPETIPET is another widely used method for inferring a global WCET estimate of a complete program from local WCET estimates of individual program parts. It was developed independently by Puschner and Schedl [PS97], and Li and Malik [LM97]. The name of the method refers to the idea of using an implicit description of the possible sequences of program parts, or paths, instead of enumerating each path individually. Despite this very general etymological roots, the term IPET is usually used to designate a specific variant where the set of allowed paths is described by integer linear constraints over the execution frequency of the individual program parts.

In this usual form of IPET, general non-linear constraints cannot be expressed directly. Such constraints must be approximated, for example, through a piecewise linear approximation.

Under the assumption of a constant local WCET estimate for each program part, a global WCET estimate can be obtained by finding a maximal solution of a weighted sum of local execution frequencies under the given constraints.

In IPET, the problem of inferring a global WCET estimate of a complete program from the local WCET estimates of its individual parts is expressed as an integer linear optimization problem: In the original variant of IPET, the execution count of the individual program parts is expressed as a set of non-negative integer variables, and the possible control flow between the individual program parts is overapproximated by flow facts that are formulated as linear constraints. The execution of each program part is associated with a constant cost, namely the local WCET estimate of that program part. The objective function of the optimization problem is the cost-weighted sum of execution counts of all program parts, and the optimal value of that function is the global WCET estimate for the complete program.

Given a CFG $(V, E, v_{start}, v_{end})$ of some program $\mathcal{P}$, we introduce, for each node $v \in V$, an integer variable f_v, called *node variable*, to capture the execution count of node v during a single execution of $\mathcal{P}$. Likewise, we introduce, for each edge $(v, w) \in E$, an integer variable $f_{(v,w)}$, called *edge variable*, to capture the number of times control is transfered from node v to node w.

It should be noted that the introduction of node variables is not strictly necessary, as all linear constraints over some node v can also be expressed as linear constraints over the sum of v's incoming (or outgoing) edges. The sole benefit of using node variables is that they can sometimes make constraints more readable.

Usually, the solution space is chosen to represent an overapproximation of all feasible execution sequences of program parts. This is established through the following connection: For any feasible execution sequence $\pi = v_1, \ldots v_n$ of $\mathcal{P}$, the valuation

$$\{f_v \mapsto c_v \mid v \in V\} \cup \{f_e \mapsto c_e \mid e \in E\}$$

must be a solution, where c_v is the number of occurrences of node v in π, and where c_e is the number of occurrences of edge e in π, i.e.,

$$c_v = |\{i \mid v_i = v, 1 \leq i \leq n\}| \quad \text{and} \quad c_e = |\{i \mid (v_i, v_{i+1}) = e, 1 \leq i < n\}|.$$

The number of occurrences of individual program parts, as well as the number of individual control-flow transitions are finite cardinal numbers and must hence be non-negative. We therefore add non-negativity constraints

$$f_v \geq 0, \text{ for all } v \in V \quad \text{and} \quad f_e \geq 0, \text{ for all } e \in E.$$

We want to model any single run of program $\mathcal{P}$. We can ensure this by requiring that there is exactly one occurrence of the start node v_{start} and exactly one occurrence of the end node v_{end}:

$$f_{v_{start}} = 1 \quad \text{and} \quad f_{v_{end}} = 1.$$

We consider only sequential programs, in which case the CFG semantics implies that each execution of some node $v \in V$ must be followed by the execution of exactly one of its immediate successor nodes $w \in V$, unless control terminates in v. Likewise, each execution of some node $v \in V$ must follow the execution of exactly one of its immediate predecessor nodes $w \in V$, unless control starts in v. This can be expressed by the linear constraints

$$f_v = \sum_{(w,v) \in E} f_{(w,v)} \text{ for all } v \in V, \qquad f_v = \sum_{(v,w) \in E} f_{(v,w)} \text{ for all } v \in V.$$

These constraints are called *structural* constraints, because they capture the static structure of the control flow encoded in the CFG, without taking into account the dynamic program behavior.

Considering the structural constraints imposed under our notion of CFG, we see that we really need only one of the previous two constraints governing v_{start} and v_{end}. Keeping both is not strictly necessary, but otherwise causes no harm.

As mentioned in Section 3.1, programs that implement the individual tasks of a real-time system always have a finite set of possible inputs vectors on which they can operate and are, moreover, required to eventually terminate on any of these inputs. Unfortunately, the CFG does not provide enough information about the control flow within its strongly connected components. Without further constraints, the control flow within cyclic regions of the CFG would be unbounded, and so would then be the whole optimization problem.

In Section 3.8, we have mentioned that many types of loop iteration constraints can be derived automatically by code analysis. In cases where this is not possible, they must be supplied by an expert who has to analyze the source code.

The most general notion of a cyclic area in a program corresponds to the notion of a strongly connected component from graph theory, i.e., there exists a path from each node inside the component to any other node inside the same component. Although it is possible to formulate loop iteration constraints for strongly connected components, for most practical programs it is sufficient to consider the more restrictive notion of a *natural loop*.

Constraints	Use Case
$f_v = k \cdot f_e$	Edge e occurs k times as often as node v. Example: Inside a loop with header v, edge e is associated with a control-flow decision that becomes true for every k-th iteration.
$f_{e_1} \leq f_{e_2}$	For each occurrence of edge e_1, there is at least one occurrence of edge e_2. Example: Inside a loop, edge e_1 is associated with a control-flow decision that implies another control-flow decision, associated with the execution of e_2, to be taken.
$f_{e_1} \leq M \cdot f_{e_2}$	If there is an occurrence of edge e_1, there is at least one occurrence of edge e_2. Constant M must have a large positive value. Example: Edge e_1 is associated with a control-flow decision that corresponds to the activation of a specific operation mode in which code guarded by the control-flow decision associated with edge e_2 is activated.
$f_{e_1} = f_{e_2}$	For each occurrence of edge e_1, there is exactly one occurrence of edge e_2. Example: The conditions associated with edge e_1 and edge e_2 guard code that is dispersed, but functionally related.
$f_{e_1} \leq M \cdot y$ $y \leq 1$	If there is an occurrence of edge e_1, there is at least one occurrence of edge e_2. Constant M must have a large positive value. Example: Edge e_1 is associated with a control-flow decision that corresponds to the activation of a specific operation mode in which code guarded by the control-flow decision associated with edge e_2 is activated.

Table 3.1: Examples of different types of constraints in IPET.

We recall the definition of a natural loop: A node v is said to *dominate* another node w, if any path from the start node v_{start} to w must pass through v. A natural loop is defined by its *back edge* (w, v), which is an edge, such that its target node v *dominates* its source node w.

Iteration constraints for such loops are typically inequalities that bound the flow through the back edge relative to the total outside flow $f_{(w_1,v)} + \ldots + f_{(w_n,v)}$ into the loop header v, i.e.,

$$f_{(w,v)} \leq b \cdot (f_{(w_1,v)} + \ldots + f_{(w_n,v)}),$$

where b is a positive integer constant, where $\{w_1, \ldots, w_n, w\}$ is the set of immediate predecessors of node v and where w dominates v.

One particular benefit of IPET is the possibility to include supplementary constraints on the control flow in the form of arbitrary linear constraints, as might be provided by program analysis or expert insight into the behavior of the underlying program. Doing so may considerably reduce the solution space, resulting in closer WCET estimates. A number of examples of additional forms of constraints are given in Table 3.1, with more to be found in [PS97].

Lastly, the objective function of a standard IPET problem takes the form

$$\sum_{v \in V} \widetilde{wcet}_v \cdot f_v,$$

where $\widetilde{wcet}_v$ is the local WCET estimate of node v. Maximizing this function yields a global WCET estimate for the whole program.

Example 3.2. Reconsider the CFG from Example 3.1. Assume that the loop can reiterate at most 7 times whenever it is entered. Furthermore, assume that WCET estimates for nodes v_1, v_2, and v_3 are $50, 20$, and 30 microseconds. We obtain the following IPET problem:

- Non-negativity constraints:

$$f_{v_{start}} \geq 0; \qquad f_{v_1} \geq 0; \qquad f_{v_2} \geq 0; \qquad f_{v_3} \geq 0; \qquad f_{v_{end}} \geq 0.$$

- Structural constraints and single run constraints:

$$\begin{aligned} f_{v_{start}} &= 1; & f_{v_{start}} &= f_{(v_{start},v_1)}; \\ f_{v_1} &= f_{(v_{start},v_1)}; & f_{v_1} &= f_{(v_1,v_2)} + f_{(v_1,v_3)}; \\ f_{v_2} &= f_{(v_1,v_2)}; & f_{v_2} &= f_{(v_2,v_3)}; \\ f_{v_3} &= f_{(v_1,v_3)} + f_{(v_2,v_3)} + f_{(v_3,v_3)}; & f_{v_3} &= f_{(v_3,v_3)} + f_{v_{end}}; \\ f_{v_{end}} &= (v_3, v_{end}); & f_{v_{end}} &= 1. \end{aligned}$$

- Iteration constraint:

$$f_{(v_3,v_3)} \leq 7 \cdot (f_{(v_1,v_3)} + f_{(v_2,v_3)}).$$

- We may add supplementary linear constraints that might have been provided by program analysis or expert insight into the behavior of the underlying program. As a made-up example, we might observe that the number of reiterations of the loop is constrained to 3 whenever the loop is entered through edge (v_2, v_3), for example, due to some program variables being set to a certain value in node v_2. We would then add an extra constraint

$$f_{(v_3,v_3)} \leq 3 \cdot f_{(v_2,v_3)}.$$

- Objective function:

$$0 \cdot f_{v_{start}} + 50 \cdot f_{v_1} + 20 \cdot f_{v_2} + 30 \cdot f_{v_3} + 0 \cdot f_{v_{end}}.$$

One particular limitation of standard IPET is the assumption of a constant local WCET estimate for each program part. This assumption turns out to be prohibitively pessimistic for the case of contemporary hardware platforms, where individual instructions can experience a considerable jitter in execution time, depending on the state of

caches, pipelines, an other hardware components (cf. Sections 3.2 and 3.3). To overcome this limitation, authors have proposed various different variants of standard IPET:

Li et al. have presented an approach where potential instruction cache misses are modeled by considering the feasible control-flow paths between code that maps to the same cache line. This allows the formulation of conservative constraints on the number of cache misses, which are modeled by introducing an additional non-negative integer variable for each program part. They demonstrate their approach on both, direct-mapped [LMW95, LMW99] and set-associative caches [LMW96].

Ottoson and Sjödin [OS97] have proposed the modeling of pipeline effects by introducing separate variables for pairs of adjacent program parts. Engblom and Ermedahl [EE99] have proposed a similar approach for larger sequences of instructions.

3.13 The Measurement-based Approach

In *Measurement-based timing analysis* (MBTA), the execution times of individual program parts are measured while the program is being executed on the intended target hardware. The observed local execution times of the individual program parts are subsequently combined into a global WCET estimate for the program as a whole, using any of the calculation methods discussed in Sections 3.9 through 3.12.

In Section 3.3, we have discussed that any method for precisely analyzing the WCET of a given piece of software on modern target hardware must be capable of dealing with both, the functional computer state and the TRCS. In MBTA, we apply advanced input-data generation techniques (cf. Section 3.14) in order to achieve a high TRCS coverage. MBTA here relies on a best-effort argument that cannot guarantee the coverage of all possible temporal behaviors. However, considering the use cases of MBTA that we have given in Section 1.5, such a best-effort approach is justified.

On the other hand, the functional computer state is handled in the same way as in static analysis, i.e., by control-flow analysis (cf. Section 3.8). Calculation methods like path-based estimate calculation (cf. Section 3.10), tree-based estimate calculation (cf. Section 3.11), or IPET (cf. Section 3.12) are then used to infer a global WCET estimate for the program as a whole. The major difference to static analysis is that the local WCET estimates originate from measurements.

In this Section we discuss the dataflow of MBTA, as depicted in Figure 3.7. The diagram can also be understood as depicting sequential processing, i.e., a workflow. The reason why it is useful to think in terms of dataflow, rather than in terms of a workflow, should become clearer in Chapter 6, where we present the extended dataflow of the FORTAS approach, which contains cycles and allows for the parallelization and pipelining of the various analysis operations.

We first take a look at the pieces of data (cf. Figure 3.7) that MBTA operates on:

Program code: This is the code of the task under analysis. Some implementations of MBTA are built to analyze machine code [BCP02, BCP03, SM10], whereas others, including the FORTAS tool, operate on the source-code level [KWRP05, WKRP09]. We have already discussed the advantages and disadvantages of either choice in Section 3.6.

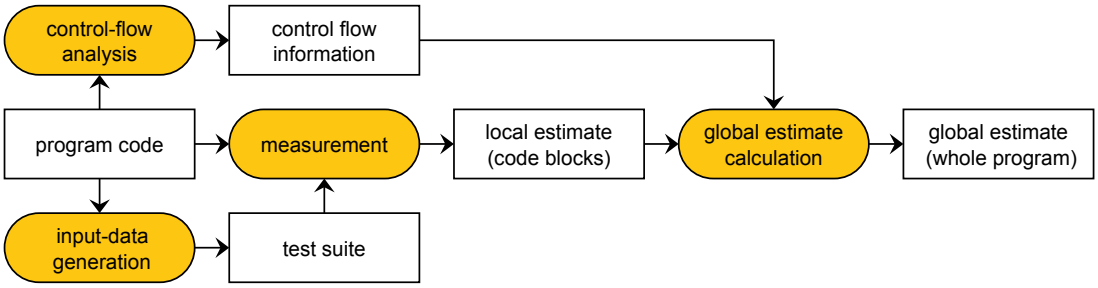

Figure 3.7: The dataflow of MBTA.

Control-flow information: Also known as flow facts, control-flow information comprises the available constraints over the feasible control flow within the software under analysis, i.e., all the information obtained during control-flow analysis (cf. Section 3.8). This includes constraints imposed by the static structure of the code, as described by the program's CFG or AST, loop bounds, and any general flow constraints that help to exclude infeasible paths.

Test suite: A test suite is a collection of input vectors that are used to initialize the computer state before executing the program for the purpose of performing a measurement. Ideally, each input vector should provoke a particular TRCS (cf. Section 3.3). In practice, it is not possible to guarantee a full coverage of the TRCS, since the exact connection between input-data space and the TRCS space is—intentionally—not modeled explicitly in MBTA. The specific type of used input vectors limits the TRCS space coverage that is achievable by a particular MBTA implementation. The term *test suite* is taken from functional testing.

Local estimates: The local WCET estimates are WCET estimates for the individual program parts. A straightforward choice are the local maximal observed execution times (MOETs). Optionally, a safety margin can be added to each local maximal observed execution time, or extreme value theory [Col01] can be applied to the collected samples.

Global estimate: The global WCET estimate is the final result of timing analysis. It approximates the value of the actual WCET of the software under analysis on the given target hardware.

Next, we consider the individual operations (cf. Figure 3.7) that are performed at analysis time:

Control-flow analysis: control-flow analysis (cf. Section 3.8) is concerned with determining the feasible sequences in which the individual program parts can be executed. The central point of control-flow analysis is to exclude as many provably infeasible execution sequences as practical from further consideration during estimate calculation.

Input data generation: Input data generation (cf. 3.14) produces suitable input vectors for the subsequent execution of the software on the target hardware.

Measurement: In the measurement stage (cf. 3.15) the program is executed on the target hardware, and execution times of the individual program parts are obtained.

Global estimate calculation: Estimate calculation is concerned with combining the local WCET estimates obtained by measurement (cf. 3.15) into a global WCET estimate for the whole program, according to the information obtained by control-flow analysis (cf. Section 3.8). Estimate calculation was discussed in Section 3.9.

3.14 Input Data Generation

Any method for precisely analyzing the WCET of a given piece of software on a modern target hardware must be capable of handling timing effects that are linked to both, the functional computer state and the TRCS. Concerning the functional computer state, the analysis must consider the effect on the execution sequence of machine instructions, as well as the effect on execution times of individual instruction sequences. With respect to the TRCS, the analysis must consider the effect on execution times of individual instruction sequences.

In MBTA, the temporal behavior of the task under analysis is determined by executing the corresponding code on the actual target platform and measuring the execution time of individual program parts. The possible sequences of program parts can be modeled separately by applying control-flow analysis (cf. Section 3.8) and calculation methods like path-based estimate calculation (cf. Section 3.10), the tree-based approach (cf. Section 3.11), or IPET (cf. Section 3.12). On the other hand, information on the execution time of individual instructions must be obtained by measurement.

Since the execution time of individual instructions depends on the TRCS, performing a single measurement for each instruction, or program part, is not sufficient to obtain enough information. Moreover, the connection between a particular TRCS and the execution time it induces in a subsequently executed instruction is complex and cannot be exploited without access to a detailed temporal model of the hardware.

Performing exhaustive measurements for all possible initial states is not tractable due to the size of the state space. All that MBTA can do is to strive to achieve a high coverage of the temporal behavior of individual instructions. Various heuristics can be used to choose a suitable subset of initial states, which we will discuss shortly.

MBTA cannot directly control the complete TRCS. As we have seen in Section 3.3, the TRCS may include parts of the hidden computer state that are not directly accessible. Moreover, incomplete knowledge of the execution platform is one of the primary assumptions of MBTA. However, MBTA can control the functional computer state. When used as initial condition for executing a piece of code, elements of the functional computer state are also called *input vectors*.

To perform execution-time measurements of an instruction or a sequence of instructions, it is important to consider whether the code should be executed within its program execution context, or if it should be executed in isolation.

If the code is executed in isolation, then the initial state before execution can be controlled more directly: For example, if a given function is executed in isolation, we can directly control the value of all the function's input parameters by specifying respective arguments. However, if the same function is executed as part of a call from another

function, we have no direct control over the parameters. Rather, we can only indirectly enforce certain arguments by controlling the input data of the calling function(s).

Generating appropriate input data for the calling function to enforce some intended input for the callee function can be difficult. On the other hand, generating input data directly for the called function introduces pessimism, since in initial state for the called function is no longer constrained by its execution context.

A mix of different techniques can be used to generate a set of input vectors [BZTK11]:

Random input-data generation: Generating random data is very fast. This allows the generation of a large amount of input vectors in a short time. Random input generation can achieve a surprisingly good coverage in a short time. However, the method tends to miss rare conditions. Figure 3.8 provides a simple example of a rare case that is likely to be missed by this technique.

Concolic testing: Concolic testing [WMMR05, CGP+08, God05, SMA05] is a technique for input-data generation from functional software testing that combines concrete execution with symbolic execution [BEL75, How76]. Concolic testing uses concrete execution to find the concrete sequence of instructions for some (initially random) input date. Subsequently it uses symbolic execution to collect *path conditions*—necessary conditions over the input variables that lead to a particular sequence of instructions—along the concrete execution path, up to a certain depth. By negating one or more path conditions, a new set of constraints over the input variables is obtained. This constraint set can be solved to compute new input data. That way, concolic testing is able to quickly generate test suites that cover many different execution paths.

Model checking: Another way to generate input data is to exploit the ability of model checking [CGP00] to generate counter examples. Software model checkers like CBMC [CKL04] or Blast [BHJM07] permit the checking of reachability properties on source code. Given a particular state that should be used for measurement, a corresponding safety property can be formulated. If the safety property is violated, i.e., if the state is in fact reachable, such a model checker can provide suitable input data in the form of a counter-example.

Stochastic optimization: Stochastic optimization can sometimes do a good job at finding input data for some rare conditions, through iterative, heuristic approximation. It cannot guarantee to cover all special cases, though. The FROO method, which we will present in Section 5.2, is an example of an input-data generation method that is based on stochastic optimization.

3.15 Measurement

During measurement, the program is executed on the target hardware, and execution times for the individual program parts are recorded.

To do so, this stage may need to prepare the program code for the execution. For example, if the program code is provided as source code, it must be compiled for respective target hardware platform. If the compiler performs optimizations that affect the control

```
void foo(int x, int y) {
    if (x == C1 && y == C2) {
       [...]

       } else {
       [...]

       }
 }
```

Figure 3.8: An example of C source code for which random input-data generation does not work well: The `true`-branch of the `if`-construct is unlikely to be invoked using input data obtained by random input-data generation. The reason is that the branch condition is only met by a single valuation of variables `x` and `y` (out of 2^{64} possibilities, assuming a 32-bit architecture.

flow, then any flow information that refers to the source code must be transformed as well to match these optimizations [KP01, KP03, KPP10b]. This transformation can be either performed by the compiler [KP05], or by an external tool that mirrors the behavior of the compiler.

The former approach is simpler, but requires that the compiler either comes with native support for the transformation of flow information, or is modified accordingly. Such a modification might be practically infeasible, like in the case of a third-party compiler for which no source code is available, or too costly, like in the case of a certified compiler that would require re-certification.

The second approach of using an external transformation tool to mirror the behavior of the compiler burdens that tool to exactly mimic all the transformations that are performed by the compiler.

Eventually, the code is executed on the target hardware, using the input data from the generated test suite, which is obtained from input-data generation (cf 3.14). This yields either a set of observed execution times for each individual program part, or timed traces, where each timed trace indicates the execution sequence of individual control-flow elements and the execution duration for each entry in this sequence.

A set of timed traces can provide more useful information on the temporal behavior of a program than the corresponding sets of execution times for each individual program part: A timed trace additionally contains information on the control flow under which a specific execution time occurred, as well as information on which execution times occurred together. This information is not used in standard MBTA, but it is essential for context-sensitive IPET (cf. Chapter 4).

In order to avoid exerting a probe effect on the system, special tracing hardware can be used to perform the execution-time measurements [II03, SM10, Lau]. Otherwise, software instrumentation can be used [Pet03, WRKP05, WKRP05, BB06, RWSP07, WKRP09, BMB10].

Various techniques can be used to capture execution times:

Tracing hardware: Modern hardware usually features specialized on-chip debugging interfaces that can be used to non-intrusively capture timed traces while the software is running [Lau, BMB10, SM10].

Software instrumentation: If such an interface is not available, the technique of software instrumentation can be applied. In this case special time stamping code is inserted into the software under scrutiny. This technique is intrusive, because it introduces a probe effect into the system.

Simulation: In case a cycle-accurate simulation of the target hardware is available, it is possible to obtain timed traces via simulation.

A major strength of MBTA is its relatively low retargeting effort. Such retargeting requires the implementation of a target-specific driver that is able to perform the following actions:

Compiling and linking: To produce executable machine code for the target platform, the code must first be compiled and linked, using a target-specific compilation toolchain.

Executable download: Before the executable machine code can be run, it must be downloaded to the target platform. This is usually done via a standard IEEE 1149.1 JTAG debug port.

Starting and stopping: After the executable machine code has been downloaded to the target platform, its execution must be started. Likewise, there must be a mechanism to stop the execution after the last instruction of the program has been processed.

Collecting timed traces: When collecting execution times, care must be taken not to exert a probe effect on the system. Ideally, the timing information should be captured using the features provided by modern, non-intrusive on-chip debugging hardware [II03, SM10, Lau]. If such features are not available, software instrumentation can be used as a fall-back technique [Pet03, WRKP05, WKRP05, BB06, RWSP07, WKRP09, BMB10]. Moreover, some preprocessing is required to make the timed traces accessible to the analysis. In particular, it might be necessary to map the traces from the executable binary level to the source-code level.

Apart from implementing a target-specific measurement driver, it is necessary to parametrize the analysis with respect to the processor's specific data-types, for example, value ranges and binary representations. The algorithms for control-flow analysis, input-data generation, and estimate calculation need not be changed, however.

3.16 Chapter Summary

In this chapter, we have discussed the preliminaries of WCET analysis.

The traditional approach for WCET analysis follows the idea of splitting the program under analysis into smaller parts and determining local WCET estimates for these individual parts. Processor-behavior analysis is concerned with determining WCET estimates for the individual program parts. Control-Flow Analysis is concerned with determining the feasible sequences in which the individual program parts can be executed. Estimate calculation is concerned with combining the local WCET estimates obtained

by processor-behavior analysis into a global WCET estimate for the whole program. In measurement-based timing analysis (MBTA) the execution times of individual program parts are measured while the program is being executed on the intended target hardware.

CHAPTER 4

Reducing Pessimism

In this chapter, we develop a method for reducing pessimism in MBTA.

In Section 4.1, we consider how pessimism arises due to overapproximation of the systems behavior. In Section 4.2, we consider a monotonicity property that is exhibited by many widely used estimate calculation methods. That property is important to guarantee that our method for reducing pessimism will never increase pessimism. In Section 4.3, we recall that the state of a system can be understood as a momentary summary of the system's execution history. As a result, it is possible to use a known execution history to recover partial information about the system's internal state at a given point within a program's execution. In Section 4.4, we present context-sensitive IPET, a generic estimate calculation method that enables the reduction of pessimism by allowing the specification of separate execution scenarios for any individual program part. In Sections 4.5 through 4.9, we present a concrete instantiation of context-sensitive IPET.

4.1 Pessimism

Pessimism arises in MBTA during estimate calculation, when we infer a global WCET estimate from individual local WCET estimates without taking into account the dependencies between the execution time of different program parts: The combination of two or more local worst-case scenarios might not be feasible along any concrete run, due to logical dependencies in the control flow.

Local WCET estimates that have been obtained by measurement are affected by optimism. However, we choose to address optimism separately, in Chapter 5. In the present chapter, we therefore assume that the local WCET estimates of each node are free of optimism, i.e., each local WCET estimate is at least as high as the actual WCET, formally:

$$\widetilde{wcet}(v) \geq wcet(v).$$

The methods for estimate calculation that we have discussed in the previous chapter—path-based estimate calculation (cf. Section 3.10), tree-based estimate calculation (cf. Section 3.11), and IPET (cf. Section 3.12)—perform an over-approximative

calculation of the WCET estimate, assuming that local WCET behaviors in two or more different program parts may coincide, even if such an execution scenario has not been observed or may not at all be exhibited by the system. Figure 4.1 illustrates the idea of pessimistic estimate calculation by a simple example.

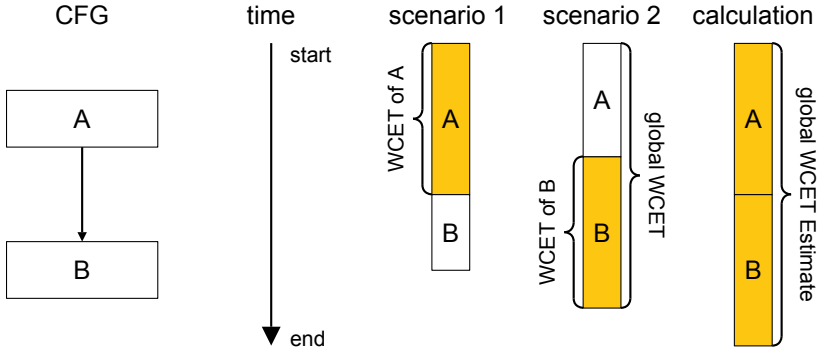

Figure 4.1: Example for pessimistic estimate calculation: In this example we assume a simple program fragment consisting of two consecutive code blocks, A and B. Furthermore, we assume that there are only two execution scenarios. Part A exhibits its WCET in the first scenario, whereas part B exhibits its WCET in second scenario. We also assume that our measurement runs cover both scenarios. A simple version of IPET will use the sum of the local WCET estimates to calculate the WCET estimate for the sequence. Using the sum, however, is pessimistic, as a run in which both local WCETs occur does not actually exist in this example. The result is a global WCET estimate that is higher than the actual global WCET. In this example, the actual global WCET shows up in Scenario 2.

The reason why estimate calculation in MBTA is inherently pessimistic is historic: these calculation techniques have been adopted from static WCET analysis. In their original context, these calculation techniques were developed with sufficient schedulability tests in mind (cf. Section 2.5). MBTA later inherited these estimate techniques without modification.

4.2 Monotonicity

The three estimate-calculation methods that we have considered—path based estimate calculation (cf. Section 3.10), tree-based estimate calculation (cf. Section 3.11), and IPET (cf. Section 3.12)—are *monotonic* in the local WCET estimates: If one of the local WCET estimates that are used for calculating the global WCET estimate increases by a positive value δ—with the other local WCET estimates staying the same—then the global WCET estimate cannot decrease. Equivalently, we formally define:

Definition 4.1 (Monotonic estimate calculation method)**.** *Let f be a function that takes as arguments local WCET estimates $\widetilde{wcet}_1, \ldots, \widetilde{wcet}_n$ and yields a global WCET estimate $\widetilde{wcet} = f(\widetilde{wcet}_1, \ldots, \widetilde{wcet}_n)$. We say that f* denotes a monotonic estimate calculation method*, iff, for any change $\delta_1, \ldots, \delta_n \geq 0$,*

$$f(\widetilde{wcet}_1 + \delta_1, \ldots, \widetilde{wcet}_n + \delta_n) \geq f(\widetilde{wcet}_1, \ldots, \widetilde{wcet}_n).$$

4.3 State as an Encoding of History

In Section 3.3, we have argued that the execution time of the individual instructions of a program ultimately depends on the computer state, specifically on the TRCS. The state of a system summarizes the information of the system's history that is relevant for its future behavior. Mesarovic and Takahara [MT89] identify the following principal reasons for the introduction of state:

> (i) A system is, in general, a relation; i.e., the same input can lead to different outputs. The state enables a representation of the system as a function. The idea is that if one knows what state the system is in, he could with assurance ascertain what the output will be. In such a way one regains "*predictability*" believed to be present if a complete set of observations is available.
>
> (ii) The state enables the determination of a future output solely on the basis of the future input and the state the system is in. In other words, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of the system. *Knowing the state supplants knowledge of the past.* Apparently, for this role to be meaningful, the notion of past and future must be relevant for the system considered; this leads to the notion of an abstract time system.

4.4 Context-sensitive IPET

Context-sensitive IPET is a generic estimate calculation method. It is an extension of standard IPET (cf. Section 3.12). Unlike standard IPET, context-sensitive IPET does not rely on a single, constant WCET estimate of each program part, but allows for multiple *execution scenarios* of each program part, with different associated WCET estimates.

Like standard IPET, context-sensitive IPET relies on solving an ILP problem. The ILP problem used by context-sensitive IPET is based on the ILP problem used by standard IPET. When switching from standard IPET to context-sensitive IPET, it is therefore possible to reuse all available IPET constraints for a given program. This is a key advantage of context-sensitive IPET over other context-sensitive calculation methods.

To construct a context-sensitive IPET problem, we reuse all the variables from the corresponding standard IPET problem: *node variables* to capture the execution count of each node $v \in V$ during a single execution of $\mathcal{P}$ and *edge variables* to capture the number of times control is transfered from node v to node w, for each edge $(v, w) \in E$. The semantics of these variables is preserved in the context-sensitive IPET problem, and we therefore keep all associated constraints from the standard IPET problem: Non-negativity constraints, constraints that enforce exactly one run, structural constrains imposed by the CFG, loop iteration constraints, as well as any supplementary constraints specified by the user or obtained through automatic analysis.

In standard IPET, we use the maximal observed execution time of each node $v \in V$ as that node's WCET estimate $\widetilde{wcet}_v$. In context-sensitive IPET, we model, for each

node $v \in V$, a set of different execution scenarios

$$E_{v,1}, \ldots, E_{v,n(v)}$$

with different associated WCET estimates

$$\widetilde{wcet}_{v,1}, \ldots, \widetilde{wcet}_{v,n(v)}.$$

The execution count of each of the execution scenarios $E_{v,1}, \ldots, E_{v,n(v)}$ of node v during a single execution of the given program $\mathcal{P}$ is modeled by respective *scenario variables*

$$f_{v,1}, \ldots, f_{v,n(v)} \quad \text{of } v.$$

The individual execution scenarios of a given node $v \in V$ should be understood as refinements of the unspecific execution scenario of v, as used in standard IPET, which is associated with the unspecific maximal observed execution time $\widetilde{wcet}_v$ in MBTA. As a consequence, the associated WCET estimates $\widetilde{wcet}_{v,1}, \ldots, \widetilde{wcet}_{v,n(v)}$ of the specific execution scenarios must not exceed $\widetilde{wcet}_v$. Hence, we require:

Requirement 4.1 (Execution scenario specialization)**.** *The WCET estimates under a specific execution scenario of some node v must not exceed the unspecific WCET estimate $\widetilde{wcet}_v$ of v, i.e.,*

$$\widetilde{wcet}_{v,i} \leq \widetilde{wcet}_v, \text{ for all } 1 \leq i \leq n(v).$$

Furthermore, we require that the execution scenarios $E_{v,1}, \ldots, E_{v,n(v)}$ of each node $v \in V$ classify the executions of v, i.e., we require that each execution of node v during a run of program $\mathcal{P}$ is associated with exactly one of the execution scenarios of v:

Requirement 4.2 (Execution scenario classification)**.** *For each node $v \in V$, the associated execution scenarios $E_{v,1}, \ldots E_{v,n(v)}$ must form a classification of all occurrences of v in any end-to-end path through the CFG.*

Adopting of Requirement 4.1 allows us to add additional constraints to the IPET problem:

$$f_v = \sum_{i=1}^{n(v)} f_{v,i}, \text{ for all } v \in V.$$

Like any other variables representing counts of occurrences, scenario variables must not be negative. We therefore include non-negativity constraints

$$f_{v,i} \geq 0, \text{ for all } v \in V, 1 \leq i \leq n(v).$$

The objective function of the context-sensitive IPET problem is a refinement of the objective function of the corresponding standard IPET problem. It is obtained by replacing each summand

$$\widetilde{wcet}_v \cdot f_v$$

by the weighted sum

$$\sum_{i=1}^{n(v)} \widetilde{wcet}_{v,i} \cdot f_{v,i},$$

i.e., the objective function of the context-sensitive IPET problem has the form

$$\sum_{v \in V} \sum_{i=1}^{n(v)} \widetilde{wcet}_{v,i} \cdot f_{v,i},$$

As in standard IPET, maximizing this objective function yields the global WCET estimate for the whole program.

Since we keep all the constraints from context-free IPET, and by way of Requirements 4.1 and 4.2, context-sensitive IPET can never be more pessimistic than context-free IPET, as stated in the following theorem:

Theorem 4.1. *The global WCET estimate obtained from a context-sensitive IPET problem never exceeds the global WCET estimate obtained from the respective standard IPET problem.*

Proof. Let $f_{v,1}, \ldots, f_{v,n(v)}$, for all $v \in V$, be a solution of the context-sensitive IPET problem. We have $f_v = \sum_{i=1}^{n(v)} f_{v,i}$ and $\widetilde{wcet}_{v,i} \leq \widetilde{wcet}_v$ for $1 \leq i \leq n(v)$, hence

$$\sum_{v \in V} \sum_{i=1}^{n(v)} \widetilde{wcet}_{v,i} \cdot f_{v,i} \leq \sum_{v \in V} \sum_{i=1}^{n(v)} \widetilde{wcet}_v \cdot f_{v,i} = \sum_{v \in V} \widetilde{wcet}_v \cdot f_v.$$

□

More interestingly, we can actually reduce pessimism in context-sensitive IPET by providing additional constraints over the scenario variables $f_{v,1}, \ldots, f_{v,n(v)}$. For example, any constraint that tightens the lower bound of some variable $f_{v,i}$ with

$$\widetilde{wcet}_{v,i} < \widetilde{wcet}_v$$

reduces the global WCET estimate, for any solution where $f_v > 0$. The classification of occurrences of a given node $v \in V$ must be performed in a way that allows such a reduction of pessimism. This is captured by the following requirement:

Requirement 4.3 (Constraints)**.** *The classification of the occurrences of a given program node $v \in V$ during a run into execution scenarios $E_{v,1}, \ldots E_{v,n(v)}$ must be performed in such a way that additional constraints over the corresponding scenario variables $f_{v,1}, \ldots, f_{v,n(v)}$ can be established.*

Example 4.1. Reconsider the IPET problem from Example 3.2. Assume that the WCET of node v_3 varies, depending on which of the three node v_1, v_2, and v_3 were executed just before node v_3. This could, for example, happen as a result of the processor's caching of accesses to instruction and data memory. Assume that the WCET estimate of node v_1, v_2, and v_3 is, respectively, 30, 10, and 5 microseconds. We therefore introduce different execution scenarios E_1, E_2, and E_3 for node v_3 and add the following constraints to the original IPET problem:

$$f_{v_3} = f_{v_3,1} + f_{v_3,2} + f_{v_3,3}.$$

The objective function of our context-sensitive IPET problem is:

$$0 \cdot f_{v_{start}} + 50 \cdot f_{v_1} + 20 \cdot f_{v_2} + 30 \cdot f_{v_3,1} + 10 \cdot f_{v_3,2} + 5 \cdot f_{v_3,3} + 0 \cdot f_{v_{end}}.$$

The context-sensitive IPET problem does not have any benefit over the standard IPET problem, unless we add some effective constraints over the individual execution scenarios. Since the execution scenario of node v_3 depends on which of its immediate predecessors v_1, v_2, or v_3 was executed immediately before, we add the following constraints:

$$f_{v_3,1} = f_{(v_1,v_3)}; \qquad f_{v_3,2} = f_{(v_2,v_3)}; \qquad f_{v_3,3} = f_{(v_3,v_3)}.$$

In this example, the total number of occurrences of edges (v_1, v_3) and (v_2, v_3) in a single run is must be 1, due to the structural constraints of the IPET problem. Therefore, adding the mentioned constraints closens the WCET estimate by enforcing a closer upper bound on $f_{v_3,3}$.

This completes our presentation of context-sensitive IPET. However, since context-sensitive IPET is a generic method, it cannot be applied right away, but must be instantiated. Like that, different concrete methods with different properties can be created. In the following sections we develop one possible instantiation of context-sensitive IPET:

- In Section 4.5, we introduce the notion of a *context* of a node, which we use to instantiate the notion of an execution scenario. We also introduce two operations that can be used to refine a given context.
- In Section 4.6, we show how we can infer context-specific linear constraints over the number of times the given node may appear on any end-to-end paths through the CFG. These constraints will help us to fulfill Requirement 4.3.
- In Section 4.7, we show how we can infer WCET estimates for individual contexts from execution times that have been obtained from measurements. The specific way estimates are assigned will allow us to fulfill Requirement 4.1.
- In Section 4.8, we present an algorithm for obtaining a set of contexts for a given node. The set of contexts is constructed in a such a way that Requirements 4.2 and 4.3 can be fulfilled.
- In Section 4.9, we show how the results can be put together to obtain a concrete instantiation of context-sensitive IPET.

4.5 Contexts

In this section we first introduce the notion of a *clip*, which is a specification of a set of paths in a CFG leading from a specific set of *entry edges* to a specific set of *exit edges*. A clip thus allows us to capture a set of paths that share a similar control flow.

The notion of a clip is the basis for the notion of a *context*. A context of a given node v is a clip where v does not occur more than once on any of the paths associated paths

of the clip. This enables a context to be used to exactly pinpoint a particular occurrence of a given node.

In Section 4.4, we have imposed Requirement 4.2, which demands that the associated execution scenarios of a given node v must form a classification of all occurrences of v in any end-to-end path through the CFG. Our plan is to use contexts as execution scenarios, so we must make sure that the contexts we use for a given node v form such a classification.

By definition, a classification must ensure that each element belongs to a class, and that no element belongs to more than one class. In other words, the set of classes must *cover* all elements, any two classes must not overlap. Since our contexts will act as classes, we present appropriate notions of coverage and disjointness.

We first present the notion of *flow coverage*, which catches the idea of capturing all possible control flows that a given node can be involved in. Then, we present the notion of *divergence*, which captures the idea of two contexts representing disjoint execution scenarios.

Next, we introduce the *simple-history context* of a node, which is a particularly simple context. The simple-history context of a node can easily be constructed from the CFG, covers the node, and can form the basis for a subsequent refinement with respect to the control-flow history.

Refinement can be achieved by splitting the initial context into smaller and smaller contexts. We conclude the section with the introduction of two important splitting operations: vertical and horizontal context splitting.

In the following, we assume a fixed CFG

$$\mathcal{G}_{\mathcal{P}} = (V, E, v_{start}, v_{end}).$$

of given program $\mathcal{P}$ of interest. Moreover, we write $\mathcal{U}$ to denote the set of end-to-end paths of $\mathcal{G}_{\mathcal{P}}$, i.e.,

$$\mathcal{U} = \{v_1 \dots v_n \mid v_1 = v_{start}, v_n = v_{end}, (v_i, v_{i+1}) \in E, 1 \leq i < n, n \in \mathbb{N}\}.$$

A *clip* is a specification of a set of paths in the CFG, leading from a specific set of *entry edges* to a specific set of *exit edges*:

Definition 4.2 (Clip)**.** *A* clip $\mathcal{S}$ *is a pair* $[\![A, B]\!]$ *consisting of a set* $A \subseteq E$ *of edges called* entry edges*, and a set* $B \subseteq E$ *of edges called* exit edges*.*

Semantically, a given clip captures a particular set of paths that share a similar control flow, starting from any of its entry edges, and ending in any of its exit edges:

Definition 4.3 (Paths in a clip)**.** *The set* $paths(\mathcal{S})$ *of* paths in a clip $\mathcal{S} = [\![A, B]\!]$ *is the set of all CFG paths that start with some entry edge in A, that end with some exit edge in B, and that do not contain any further entry or exit edges in between, i.e.,*

$$\begin{aligned} paths(\mathcal{S}) = \{v_1 \dots v_n \mid\ & (v_1, v_2) \in A, (v_{n-1}, v_n) \in B, \\ & (v_i, v_{i+1}) \in E \setminus (A \cup B), 1 < i < n-1, n \geq 3\}. \end{aligned}$$

Note, that this definition implies that each path in a clip must have at least two edges (three nodes).

NICETOHAVE: Figure

Example 4.2. Reconsider the CFG from Example 3.1. Consider the clip

$$\mathcal{S}_1 = [\![\{(v_1, v_2)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!].$$

The set of paths in $\mathcal{S}_1$ is

$$paths(\mathcal{S}_1) = \{v_1v_2v_3v_3, v_1v_2v_3v_{end}\}.$$

Note, that $paths(\mathcal{S}_1)$ does not contain any longer paths, because the back edge of the loop is an exit edge.

A *context* of a given node $v \in V$ is a clip $\mathcal{S}$, such that v may occur at most once *inside* any path of $\mathcal{S}$. By the *inside of a path*, we mean anything *between* its first and its last node, but we do not include these border nodes themselves. The idea behind this constraint is that a context of a node v should make it possible to exactly pinpoint a specific occurrence of v.

Ideally, each path of $\mathcal{S}$ should contain *exactly* one occurrence of v inside of itself, but requiring this would be too restrictive. We therefore allow paths with no occurrence inside of itself, but note that any algorithm for constructing clips should try to minimize such paths. Formally, we define:

Definition 4.4 (Context)**.** *A* context $\mathcal{C}$ of a node $v \in V$ *is a clip, such that any path* $v_1 \dots v_n$ *in* $\mathcal{C}$ *contains at most one occurrence of* v *in its inner part* $v_2 \dots v_{n-1}$*, i.e.,*

$$v_1 \dots v_n \in paths(\mathcal{C}), v_i = v_j = v, 1 < i, j < n \implies i = j.$$

Example 4.3. Reconsider the CFG from Example 3.1. The clip $\mathcal{S}_1 = [\![A_1, B_1]\!]$ with

$$A_1 = \{(v_3, v_3)\}, \text{ and } \quad B_1 = \{(v_3, v_3), (v_3, v_{end})\}$$

is a context of node v_3, because none of the paths $paths(\mathcal{S}_1) = \{v_3v_3v_3, v_3v_3v_{end}\}$ in $\mathcal{S}$ contains more than one occurrence of v_3 inside itself. The clip $\mathcal{S}_2 = [\![A_2, B_2]\!]$ with

$$A_2 = \{(v_2, v_3)\}, \text{ and } \quad B_2 = \{(v_3, v_{end})\}$$

is not a context of v_3, because the path $v_2v_3v_3v_{end} \in paths(\mathcal{S}_2)$ contains two occurrences of node v_3 inside itself.

The notion of *flow coverage* catches the idea of capturing all possible control flows that a given node can be involved in:

Definition 4.5 (Flow coverage)**.** *A set of paths* X covers *a node* $v \in V$*, iff, for all non-empty paths* ρ, σ *with* $\rho \circ v \circ \sigma \in \mathcal{U}$*, there are subpaths* ρ_1, σ_2 *and non-empty subpaths* ρ_2, σ_1*, with* $\rho_1 \circ \rho_2 = \rho$ *and* $\sigma_1 \circ \sigma_2 = \sigma$*, such that* $\rho_2 \circ v \circ \sigma_1 \in X$*.*

Example 4.4. Reconsider the CFG from Example 3.1. We have

$$\mathcal{U} = \{v_{start}v_1\pi v_{end}, v_{start}v_1v_2\pi v_{end} \mid \pi \in \{v_3\}^+\}.$$

The clip

$$\mathcal{S}_3 = [\![\{(v_1, v_3), (v_1, v_2), (v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!]$$

covers node v_3. The clip

$$\mathcal{S}_4 = [\![\{(v_1, v_3), (v_1, v_2)(v_3, v_3)\}, \{(v_3, v_{end})\}]\!]$$

does not cover node v_3.

The notion of *divergence* captures the idea of two contexts representing disjoint execution scenarios. We first define divergence on individual paths, and then extend the definition to sets of paths:

Definition 4.6 (Divergent paths)**.** *Two paths π and σ are* divergent*, iff π and σ do not overlap on more than a single edge, and none is a subpath of the other, i.e., none of the following apply:*

- *there exist paths α, β, γ with $\alpha \circ \beta = \pi$, $\beta \circ \gamma = \sigma$, and $|\beta| \geq 2$;*
- *there exist paths α, β, γ with $\alpha \circ \beta = \sigma$, $\beta \circ \gamma = \pi$ and $|\beta| \geq 2$;*
- *σ is a subpath of π;*
- *π is a subpath of σ.*

Theorem 4.2 (Divergence of paths in clip)**.** *Let $\mathcal{S} = [\![A, B]\!]$ be a clip, and let $\pi, \rho \in paths(\mathcal{S})$ be paths of $\mathcal{S}$, with $\pi \neq \rho$. Then π and ρ are divergent.*

Proof. Let $\mathcal{S} = [\![A, B]\!]$ be a clip. Choose any two paths

$$\pi = u_1 \dots u_n \in paths(\mathcal{S}) \quad \text{and} \quad \sigma = w_1 \dots w_m \in paths(\mathcal{S}), \quad \text{with } \pi \neq \sigma.$$

Assume that paths π and σ are not divergent. There are four cases:

Case 1: There exist paths α, β, γ with $\alpha \circ \beta = \pi$, $\beta \circ \gamma = \sigma$, and $|\beta| \geq 2$. Since $\pi \neq \sigma$, we have $|\alpha| \geq 1$ or $|\gamma| \geq 1$ (or both). If $|\alpha| \geq 1$, then π contains some entry edge $(u_i, u_{i+1}) \in A$, where $2 \leq i \leq n-2$. This contradicts the assumption that $\pi \in paths(\mathcal{S})$. If $|\gamma| \geq 1$, then σ contains some exit edge $(w_i, w_{i+1}) \in B$, where $2 \leq i \leq m-2$. This contradicts the assumption that $\sigma \in paths(\mathcal{S})$.

Case 2: There exist paths α, β, γ with $\alpha \circ \beta = \sigma$, $\beta \circ \gamma = \pi$ and $|\beta| \geq 2$. This case is symmetric to Case 1.

Case 3: Path σ is a subpath of π. Since $\sigma \neq \pi$, path σ must be a proper subpath of π, hence π contains some edge $(u_i, u_{i+1}) \in A \cup B$, where $2 \leq i \leq n-2$. This contradicts the assumption that $\pi \in paths(\mathcal{S})$.

Case 4: Path π is a subpath of σ. This case is symmetric to Case 3.

□

Definition 4.7 (Divergent sets)**.** *Two sets of paths X and Y are* divergent*, iff, for any two paths $\pi \in X$ and $\sigma \in Y$, π and σ are divergent.*

Example 4.5. Reconsider clip

$$\mathcal{S}_4 = [\![\{(v_1, v_3), (v_1, v_2)(v_3, v_3)\}, \{(v_3, v_{end})\}]\!]$$

from Example 4.4. The paths of clip $\mathcal{S}_4$ and the paths of clip

$$\mathcal{S}_5 = [\![\{(v_2, v_3)\}, \{(v_3, v_{end})\}]\!]$$

are not divergent, because the path $v_2v_3v_{end} \in paths(\mathcal{S}_5)$ is a subpath of the path $v_1v_2v_3v_{end} \in paths(\mathcal{S}_4)$. On the other hand, the paths of clip $\mathcal{S}_4$ and clip

$$\mathcal{S}_6 = [\![\{(v_1, v_3), (v_1, v_2)(v_3, v_3)\}, \{(v_3, v_3)\}]\!]$$

are divergent.

The *simple-history context* of a node is a particularly simple context. The simple-history context of a node can easily be constructed from the CFG, covers the node, and can form the basis for a subsequent refinement with respect to the control-flow history. We first define the *simple-history clip* of given node $v \in V \setminus \{v_{start}, v_{end}\}$, and then show that it is a context that covers v:

Definition 4.8 (Simple history clip)**.** *Let $v \in V \setminus \{v_{start}, v_{end}\}$, i.e., let v be a node that is neither the start node, nor the end node. The* simple-history clip *of v is the clip*

$$\mathcal{S} = [\![(Q \cup B) \cap R, B]\!],$$

where Q is the set of all edges $(v_{start}, q) \in E$ that start with the start node v_{start}; where B is the set of all outgoing edges $(v, b) \in E$ of v; where R is the set of all edges $(w, y) \in E$, such that there is a path from r to v, i.e.,

$$Q = \{(v_{start}, q) \in E\}; \quad B = \{(v, b) \in E\}; \quad R = \{(w, r) \in E \mid (r, v) \in E^*\}.$$

Theorem 4.3 (Simple history context)**.** *Let $v \in V \setminus \{v_{start}, v_{end}\}$, i.e., let v be a node that is neither the start node, nor the end node. Let Q be the set of all edges $(v_{start}, q) \in E$ that start with the start node v_{start}. Let B be the set of all outgoing edges $(v, b) \in E$ of v. Let R be the set of all edges $(w, y) \in E$, such that there is a path from r to v, i.e.,*

$$Q = \{(v_{start}, q) \in E\}; \quad B = \{(v, b) \in E\}; \quad R = \{(w, r) \in E \mid (r, v) \in E^*\}.$$

Then the simple-history clip $\mathcal{S} = [\![(Q \cup B) \cap R, B]\!]$ is a context, and $paths(\mathcal{S})$ covers v.

Proof. We first show that $\mathcal{S}$ is a context, and then show the coverage property:

1. We show that $\mathcal{S}$ is a context of node v. Consider any path $\pi \in paths(\mathcal{S})$. Any edge (v, b) with source node v is an exit edge of clip $\mathcal{S}$. Therefore, π can contain at most one more node after the first occurrence of v. Hence, π contains at most one occurrence of v inside itself.

2. We show that $\mathcal{S}$ covers v. Consider any paths ρ, σ with $\rho \circ v \circ \sigma \in \mathcal{U}$. Note that path ρ starts with the start node v_{start} and that path σ ends with the end node v_{end}, which implies that paths ρ and σ are not empty.

Case 1: Path ρ contains node v. Then choose paths τ_1, τ_2 with $\tau_1 \circ v \circ \tau_2 = \rho$, such that τ_2 does not contain node v. Path $v \circ \tau_2 \circ v$ starts with an edge in $B \cap R$, and path $\tau_2 \circ v$ contains no edge in $Q \cup B$, and path $v \circ \sigma$ starts with an edge $(v, b) \in B$. Hence, path $v \circ \tau_2 \circ v \circ b$ is in $paths(\mathcal{S})$, and there are subpaths ρ_1, σ_2 with $\rho_1 \circ v \circ \tau_2 = \rho$ and $b \circ \sigma_2 = \sigma$.

Case 2: Path ρ does not contain node v. Then path $\rho \circ v$ starts with an edge in $Q \cap R$, path $\rho \circ v$ contains no edge in B, except for its first edge $(v, b) \in B$, and path $v \circ \sigma$ starts with an edge $(v, b) \in B \cup Q$. Hence, path $\sigma \circ v \circ b$ is in $paths(\mathcal{S})$, and there are subpaths ρ_1, σ_2 with $\rho_1 \circ \rho = \rho$ and $b \circ \sigma_2 = \sigma$.

□

Example 4.6. Reconsider the CFG from Example 3.1. The clip

$$\mathcal{S} = \{(v_{start}, v_1), (v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}$$

is the simple-history context of node v_3.

We can refine contexts by splitting them. A new set of contexts for a given node $v \in V \setminus \{v_{start}, v_{end}\}$ can be obtained by recursive splitting some initial context $\mathcal{C}$, for example with the simple-history context. The used splitting operations should be flexible enough to allow the splitting of a given context into suitable subcontexts. However, such an operation should also preserve the coverage of the context it operates on: If the paths of the original context cover node v, then the paths of the resulting subcontexts should, together, cover v. Moreover, the resulting subcontexts should not overlap, i.e., their sets of paths should diverge. Like this, recursive splitting of a suitable initial context will produce a set of contexts that forms a suitable classification.

First, we consider *vertical context splitting*, which permits the isolation of a selected sets of subpaths:

Definition 4.9 (Vertical context splitting)**.** *Let $\mathcal{C} = [\![A, B]\!]$ be a context of node $v \in V$.*

Let X be a set of edges $(x_1, x_2) \in E \setminus (A \cup B)$, such that there exists an edge $(a, u) \in A$ with a path from node u to node x_1 that contains only edges in $E \setminus (A \cup B)$, and such that there exists an edge $(w, b) \in B$ with a path from node x_2 to node w that contains only edges in $E \setminus (A \cup B)$.

Let Y be the set of all edges $(w, y) \in E$, such that there exists an edge $(a, u) \in A$ with a path from node u to node w that contains only edges in $E \setminus (A \cup B \cup X)$.

Let Z be the set of all edges $(w, z) \in E$, such that there exists an edge $(x, u) \in X$ with a path from node u to node w that contains only edges in $E \setminus (A \cup B \cup X)$.

The application of vertical context splitting *to context $\mathcal{C}$ produces two new contexts*

$$\mathcal{C}_1 = [\![A, (B \cup X) \cap Y]\!] \quad \textit{and} \quad \mathcal{C}_2 = [\![X, (B \cup X) \cap Z]\!].$$

The fact that $\mathcal{C}_1$ and $\mathcal{C}_2$ are contexts is shown in the following theorem.

Theorem 4.4 (Vertical context splitting)**.** *Let $\mathcal{C} = [\![A, B]\!]$ be a context of node $v \in V$.*

Let X be a set of edges $(x_1, x_2) \in E \setminus (A \cup B)$, such that there exists an edge $(a, u) \in A$ with a path from node u to node x_1 that contains only edges in $E \setminus (A \cup B)$, and such

that there exists an edge $(w, b) \in B$ with a path from node x_2 to node w that contains only edges in $E \setminus (A \cup B)$.

Let Y be the set of all edges $(w, y) \in E$, such that there exists an edge $(a, u) \in A$ with a path from node u to node w that contains only edges in $E \setminus (A \cup B \cup X)$.

Let Z be the set of all edges $(w, z) \in E$, such that there exists an edge $(x, u) \in X$ with a path from node u to node w that contains only edges in $E \setminus (A \cup B \cup X)$.

Then the following assertions hold:

1. *$\mathcal{C}_1 = [\![A, (B \cup X) \cap Y]\!]$ and $\mathcal{C}_2 = [\![X, (B \cup X) \cap Z]\!]$ are contexts of v;*
2. *if $paths(\mathcal{C})$ covers v, then $paths(\mathcal{C}_1) \cup paths(\mathcal{C}_2)$ covers v;*
3. *$paths(\mathcal{C}_1)$ and $paths(\mathcal{C}_2)$ are divergent.*

Proof. First, we show that clips $\mathcal{C}_1$ and $\mathcal{C}_2$ are contexts of v. Next, we show that $paths(\mathcal{C}_1) \cup paths(\mathcal{C}_2)$ covers v. Lastly, we show $paths(\mathcal{C}_1)$ and $paths(\mathcal{C}_1)$ are divergent.

1. We show that $\mathcal{C}_1$ and $\mathcal{C}_2$ are contexts of v. For every edge $(x_1, x_2) \in X$, there exists an edge $(a, u) \in A$ with a path from node u to node x_1 that contains only edges in $E \setminus (A \cup B)$ and an edge $(w, b) \in B$ with a path from node x_2 to node w that contains only edges in $E \setminus (A \cup B)$. Therefore, every path in $paths([\![A, X]\!])$ is a subpath of some path in $paths([\![A, B]\!])$, hence $[\![A, X]\!]$ is a context of v.

   By a similar argument, every path in $paths([\![X, B]\!])$ is a subpath of some path in $paths([\![A, B]\!])$, hence $[\![X, B]\!]$ is a context of v.

   Lastly, every path in $paths([\![X, X]\!])$ is a subpath of some path in $paths([\![A, B]\!])$, hence $[\![X, X]\!]$ is a context of v. We have

   $$paths([\![A, (B \cup X) \cap Y]\!]) \subseteq paths([\![A, B \cup X]\!]) \subseteq paths([\![A, B]\!]) \cup paths([\![A, X]\!]);$$
   $$paths([\![X, (B \cup X) \cap Z]\!]) \subseteq paths([\![X, B \cup X]\!]) \subseteq paths([\![X, B]\!]) \cup paths([\![X, X]\!]),$$

   hence $[\![A, (B \cup X) \cap Y]\!]$ and $[\![X, (B \cup X) \cap Z]\!]$ are contexts of v.

2. We show that $paths(\mathcal{C}_1) \cup paths(\mathcal{C}_2)$ covers v. Choose any paths ρ, σ, with $\rho \circ v \circ \sigma \in \mathcal{U}$. By our initial assumption, $paths(\mathcal{C})$ covers v, i.e., there are subpaths ρ_1, σ_2 and non-empty subpaths ρ_2, σ_1, with $\rho_1 \circ \rho_2 = \rho$ and $\sigma_1 \circ \sigma_2 = \sigma$, such that $\rho_2 \circ v \circ \sigma_1 \in paths(\mathcal{C})$. Moreover, there is some entry edge $(a, u) \in A$, some exit edge $(w, b) \in B$, and paths α, β with $a \circ u \circ \alpha = \rho_2 \circ v$ and $\beta \circ w \circ b = v \circ \sigma_1$. We are going to show, by construction, that there are always subpaths ρ_1', σ_2' and non-empty subpaths ρ_2', σ_1', with $\rho_1' \circ \rho_2' = \rho$ and $\sigma_1' \circ \sigma_2' = \sigma$, such that $\rho_2' \circ v \circ \sigma_1' \in paths(\mathcal{C}_1) \cup paths(\mathcal{C}_2)$.

   Case 1: Path $u \circ \alpha \circ v \circ \beta \circ w$ does not contain any edge in X. Then $u \circ \alpha \circ v \circ \beta \circ w$ contains only edges in $E \setminus (A \cup B \cup X)$, therefore edge (w, b) is in $B \cap Y$. Hence, choose $\rho_2' = \rho_2$ and $\sigma_1' = \sigma_1$. Path $\rho_2' \circ v \circ \sigma_1'$ is in $paths(\mathcal{C}_1)$.

   Case 2: Path $u \circ \alpha \circ v$ contains some edge in X, but path $v \circ \beta \circ w$ does not. Then there is some edge $(x_1, x_2) \in X$ and paths α_1, α_2, with $\alpha_1 \circ x_1 \circ x_2 \circ \alpha_2 = u \circ \alpha \circ v$, such that path $x_2 \circ \alpha_2$ contains only edges in $E \setminus X$. Path $x_2 \circ \alpha_2 \circ \beta \circ w$ then contains only edges in $E \setminus (A \cup B \cup X)$, therefore edge (w, b) is in $B \cap Z$. Hence, choose $\rho_2' \circ v = x_1 \circ x_2 \circ \alpha_2$ and $\sigma_1' = \beta \circ w \circ b$. Path $\rho_2' \circ v \circ \sigma_1'$ is in $paths(\mathcal{C}_2)$.

Case 3: Path $u \circ \alpha \circ v$ does not contain any edge in X, but path $v \circ \beta \circ w$ does. Then there is some edge $(y_1, y_2) \in X$ and paths β_1, β_2, with $\beta_1 \circ y_1 \circ y_2 \circ \beta_2 = v \circ \beta \circ w$, such that path $\beta_1 \circ y_1$ contains only edges in $E \setminus X$. Path $u \circ \alpha \circ \beta_1 \circ y_1$ then contains only edges in $E \setminus (A \cup B \cup X)$, therefore edge (y_1, y_2) is in $X \cap Y$. Hence, choose $\rho_2' = a \circ u \circ \alpha$ and $\sigma_1 \circ v = \beta_1 \circ y_1 \circ y_2$. Path $\rho_2' \circ v \circ \sigma_1'$ is in $paths(\mathcal{C}_1)$.

Case 4: Path $u \circ \alpha \circ v$ contains some edge in X, and so does path $v \circ \beta \circ w$. Then there is some edge $(x_1, x_2) \in X$ and paths α_1, α_2, with $\alpha_1 \circ x_1 \circ x_2 \circ \alpha_2 = u \circ \alpha \circ v$, such that path $x_2 \circ \alpha_2$ contains only edges in $E \setminus X$. Also, there is some edge $(y_1, y_2) \in X$ and paths β_1, β_2, with $\beta_1 \circ y_1 \circ y_2 \circ \beta_2 = v \circ \beta \circ w$, such that path $\beta_1 \circ y_1$ contains only edges in $E \setminus X$. Now choose α_3 with $\alpha_3 \circ v = x_2 \circ \alpha_2$, and choose β_3 with $v \circ \beta_3 = \beta_2 \circ y_1$. Path $\alpha_3 \circ v \circ \beta_3$ then contains only edges in $E \setminus (A \cup B \cup X)$, therefore edge (y_1, y_2) is in $X \cap Z$. Choose $\rho_2' \circ v = x_1 \circ x_2 \circ \alpha_2$ and $v \circ \sigma_1' = \beta_2 \circ y_1 \circ y_2$. Path $\rho_2' \circ v \circ \sigma_1'$ is in $paths(\mathcal{C}_2)$.

3. We show that $paths(\mathcal{C}_1)$ and $paths(\mathcal{C}_2)$ are divergent.

   Consider the paths

   $$\pi = u_1 \dots u_n \in paths(\mathcal{C}_1) \quad \text{and} \quad \sigma = w_1 \dots w_m \in paths(\mathcal{C}_2).$$

   There are two cases how π and σ may overlap:

   Case 1: π contains the first edge $(w_1, w_2) \in X$ of σ. Since $X \subseteq E \setminus (A \cup B)$ and $(u_1, u_2) \in A$, we therefore have $(u_i, u_{i+1}) \in X$, for some i with $2 \leq i \leq (n-1)$. However, by the definition of Y, that means $(u_{n-1}, u_n) \notin Y$, unless $i = n-1$. Hence, any occurrence of the first edge (w_1, w_2) of π must be on last edge (u_{n-1}, u_n) of σ. Since π is a path of a context, it contains at least two edges—an entry edge and an exit edge. Therefore, π cannot be a subpath of σ, and there are no paths α, β, γ with $\alpha \circ \beta = \pi$, $\beta \circ \gamma = \sigma$, and $|\beta| \geq 2$.

   Case 2: σ contains the first edge $(u_1, u_2) \in A$ of π. Since $X \subseteq E \setminus (A \cup B)$ and $(w_1, w_2) \in X$, we therefore have $(w_i, w_{i+1}) \in A$, for some i with $2 \leq i \leq (m-1)$. However, by the definition of Z, that means $(w_{m-1}, u_m) \notin Z$, unless $i = m-1$. Hence, any occurrence of the first edge (u_1, u_2) of σ must be on last edge (w_{m-1}, w_m) of π. Since σ is a path of a context, it contains at least two edges—an entry edge and an exit edge. Therefore, σ cannot be a subpath of π, and there are no paths α, β, γ with $\alpha \circ \beta = \sigma$, $\beta \circ \gamma = \pi$, and $|\beta| \geq 2$.

□

Example 4.7. Reconsider the simple-history context

$$\mathcal{C}_{v_3} = \mathcal{S} = \{(v_{start}, v_1), (v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}$$

of node v_3 from Example 4.6. There is a path from node $v_1 \in A$ to node v_1 that contains only edges in $E \setminus \{(v_{start}, v_1), (v_3, v_3), (v_3, v_{end})\}$, and there is a path from node v_2 to node v_3 that contains only edges from $E \setminus \{(v_{start}, v_1), (v_3, v_3), (v_3, v_{end})\}$, so we may choose

$$X = \{(v_1, v_2)\}.$$

Hence, we have

$$E \setminus (A \cup B \cup X) = \{(v_1, v_3), (v_2, v_3)\};$$
$$Y = \{(v_1, v_2), (v_1, v_3), (v_3, v_3), (v_3, v_{end})\};$$
$$Z = \{(v_2, v_3), (v_3, v_3), (v_3, v_{end})\}.$$

Therefore, we obtain subcontexts

$$\mathcal{C}_{v_3,1.0} = [\![A, (A \cup B \cup X) \cap Y]\!] = [\![\{(v_{start}, v_1), (v_3, v_3)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,2.0} = [\![X, (A \cup B \cup X) \cap Z]\!] = [\![\{(v_1, v_2)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!].$$

Whereas vertical context splitting permits the isolation of a selected set of subpaths, *horizontal context splitting* permits the isolation of a selected subset of paths:

Definition 4.10 (Horizontal context splitting)**.** *Let $\mathcal{C} = [\![A, B]\!]$ be a context of node $v \in V$, and let $\mathcal{D}$ be a partition of A. For each set $D \in \mathcal{D}$, let Z_D be the set of all edges $(u, w) \in E$, such that there exists some edge $(d, x) \in D$ with a path from node x to node u that contains only edges in $E \setminus (A \cup B)$. The application of* horizontal context splitting *to context $\mathcal{C}$ produces the set of contexts $\{\mathcal{C}_D \mid D \in \mathcal{D}\}$ of v, where*

$$\mathcal{C}_D = [\![D, B \cap Z_D]\!].$$

The fact that $\mathcal{C}_D$ is a context of v, for any $D \in \mathcal{D}$, is shown in the following theorem.

Theorem 4.5 (Horizontal context splitting)**.** *Let $\mathcal{C} = [\![A, B]\!]$ be a context of node $v \in V$, and let $\mathcal{D}$ be a partition of A. For each set $D \in \mathcal{D}$, let Z_D be the set of all edges $(u, w) \in E$, such that there exists some edge $(d, x) \in D$ with a path from node x to node u that contains only edges in $E \setminus (A \cup B)$. Then $\mathcal{C}_D = [\![D, B \cap Z_D]\!]$ is a context of v. Moreover, the sets $paths(\mathcal{C}_{D_1})$ and $paths(\mathcal{C}_{D_2})$, are divergent, for any sets $D_1, D_2 \in \mathcal{D}$ with $D_1 \neq D_2$. Furthermore, if $W \cup paths(\mathcal{C})$ covers node v, for any set of paths W, then $W \cup \bigcup_{D \in \mathcal{D}} paths(\mathcal{C}_D)$ covers node v.*

Proof. First, we show that clip $\mathcal{C}_D$ is a context of node v, for any $D \in \mathcal{D}$. Next, we show that $W \cup \bigcup_{D \in \mathcal{D}} paths(\mathcal{C}_D)$ covers node v, if $W \cup paths(\mathcal{C})$ covers v, for any set of paths W. Lastly, we show that $paths(\mathcal{C}_{D_1})$ and $paths(\mathcal{C}_{D_2})$ are divergent, for any sets $D_1, D_2 \in \mathcal{D}$ with $D_1 \neq D_2$.

1. We show that clip $\mathcal{C}_D$ is a context of node v, for any $D \in \mathcal{D}$. We have $D \subseteq A$, because $\mathcal{D}$ is a partition of A, and $B \cap Z_D \subseteq B$. Therefore, $paths(\mathcal{C}_D)$ is a subset of $paths(\mathcal{C})$.

   Clip $\mathcal{C}$ is a context of node v, i.e., all paths in $paths(\mathcal{C})$ contain at most one occurrence of v. Since $paths(\mathcal{C}_D)$ is a subset of $paths(\mathcal{C})$, its paths also contain at most one occurrence of v. It follows that $\mathcal{C}_D$ is a context of v.

2. We show that $W \cup \bigcup_{D \in \mathcal{D}} paths(\mathcal{C}_D)$ covers node v, if $W \cup paths(\mathcal{C})$ covers v, for any set of paths W. Choose any non-empty paths ρ, σ, with $\rho \circ v \circ \sigma \in \mathcal{U}$. By assumption, $W \cup paths(\mathcal{C})$ covers v, i.e., there are subpaths ρ_1, σ_2 and non-empty subpaths ρ_2, σ_1, with $\rho_1 \circ \rho_2 = \rho$ and $\sigma_1 \circ \sigma_2 = \sigma$, such that $\rho_2 \circ v \circ \sigma_1 \in W \cup paths(\mathcal{C})$.

If $\rho_2 \circ v \circ \sigma_1 \in W$, then $\rho_2 \circ v \circ \sigma_1 \in W \cup \bigcup_{D \in \mathcal{D}} paths(\mathcal{C}_D)$, and we are done.

Otherwise, path ρ_2 starts with some entry edge $(a, x) \in A$ of context $\mathcal{C}$, and since $\mathcal{D}$ is a partition of A, there is some set $D \in \mathcal{D}$, such that $(a, x) \in D$. Furthermore, path σ_1 ends with some exit edge $(y, b) \in B$ of context $\mathcal{C}$, and there are no further occurrence of any edge from $A \cup B$ in $\rho_2 \circ v \circ \sigma_1$, therefore (y, b) is in $B \cap Z_D$, hence $\rho_2 \circ v \circ \sigma_1 \in paths(\mathcal{C}_D) \subseteq W \cup \bigcup_{D \in \mathcal{D}} paths(\mathcal{C}_D)$.

3. We show that $paths(\mathcal{C}_{D_1})$ and $paths(\mathcal{C}_{D_2})$ are divergent, for any sets $D_1, D_2 \in \mathcal{D}$ with $D_1 \neq D_2$. Choose any path $\pi \in paths(\mathcal{C}_{D_1})$, and any path $\sigma \in paths(\mathcal{C}_{D_2})$. Since $D_1 \cap D_2 = \emptyset$, paths π and σ must have a different entry edge.

   Moreover, we have $D_1 \subseteq A$ and $\sigma \in paths(\mathcal{C})$, therefore the entry edge of π can only occur on the last edge of σ, hence π cannot be a subpath of σ, and there are no paths α, β, γ with $\alpha \circ \beta = \pi$, $\beta \circ \gamma = \sigma$, and $|\beta| \geq 2$.

   Likewise, we have $D_2 \subseteq A$ and $\pi \in paths(\mathcal{C})$, therefore the entry edge of σ can only occur on the last edge of π, hence σ cannot be a subpath of π, and there are no paths α, β, γ with $\alpha \circ \beta = \sigma$, $\beta \circ \gamma = \pi$, and $|\beta| \geq 2$.

   We conclude that $\mathcal{C}_{D_1}$ and $\mathcal{C}_{D_1}$ are divergent.

□

Example 4.8. Reconsider context

$$\mathcal{C}_{v_3,1.0} = [\![\{(v_{start}, v_1), (v_3, v_3)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!]$$

from Example 4.7. Choose the following partition of A:

$$\mathcal{D} = \{\{(v_{start}, v_1)\}, \{(v_3, v_3)\}\}.$$

We obtain the two new contexts of v_3:

$$\mathcal{C}_{v_3,1.1} = [\![\{(v_{start}, v_1)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,1.2} = [\![\{(v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!].$$

4.6 Context Constraints

A *context* of a given node $v \in V$ is a clip $\mathcal{S}$, such that v may occur at most once inside any path of S. The idea behind this constraint is that a context of a node v should make it possible to exactly pinpoint a specific occurrence of v. The next step then is to, in turn, relate the occurrence of a given context $\mathcal{C} = [\![A, B]\!]$ to the occurrence of its entry edges A and its exit edges B. Both relations are established through linear constraints that we will later translate to IPET constraints. To do all this, we must first formalize the various notions of *occurrence* that we are referring about.

First, we define the notion of an occurrence of a node $v \in V$ in a path. Also, we define the notion of an occurrence of an edge $e \in E$ in a path. Both of these notions are rather straightforward formalizations of the natural concept of an occurrence.

Next, we define the slightly more elaborate notion of a *covered occurrence* of a node $v \in V$ in a path, which is conceptually similar to the notion of an occurrence of a node in a path, but only considers occurrences of v that occur inside a path of a given clip $\mathcal{S}$.

The next step is to relate the occurrence of a given node $v \in V$ to the occurrence of its contexts, using linear constraints that we will later translate to IPET constraints.

As a last step, we relate the occurrences of a given context $\mathcal{C} = [\![A, B]\!]$ to the occurrence of its entry edge A and its exit edges B, using, again, linear constraints that we will later translate IPET constraints.

For more convenience in the rest of our presentation, we first define the following concatenation operator on paths and edges:

Definition 4.11 (Concatenation operator)**.** *The* concatenation operator $\circ$ *is defined through the following equations:*

$$\begin{aligned} v_1 \dots v_n \circ v_1 \dots v_m &= v_1 \dots v_n v_1 \dots v_m && \textit{(concatenation of two paths)} \\ v_1 \dots v_n \circ (v_a, v_b) &= v_1 \dots v_n v_a v_b && \textit{(concatenation of path and edge)} \\ (v_a, v_b) \circ v_1 \dots v_n &= v_a v_b v_1 \dots v_n && \textit{(concatenation of edge and path)} \\ (v_a, v_b) \circ (v_c, v_d) \dots v_n &= v_a v_b v_c v_d && \textit{(concatenation of two edges)} \end{aligned}$$

We define the set of occurrences of a node $v \in V$ in a path as follows:

Definition 4.12 (Occurrences of a node)**.** *The* set $occ(v, \pi)$ of occurrences of a node $v \in V$ in a path π *is defined as*

$$occ(v, \pi) = \{(\rho, \sigma) \mid \rho \circ v \circ \sigma = \pi\}.$$

Example 4.9. Consider path $\pi = v_{start} v_1 v_3 v_3 v_3 v_{end}$. We have

$$occ(v_3, \pi) = \{(v_{start} v_1, v_3 v_3 v_{end}), (v_{start} v_1 v_3, v_3 v_{end}), (v_{start} v_1 v_3 v_3, v_{end})\}.$$

Besides giving us an easy way to count the number of occurrences of a given node in a given path, Definition 4.12 affords us the following equality, which is very useful in situations where we need to refer to occurrences in different paths:

Theorem 4.6. *Let $v \in V$ be a node, and let X be a set of paths. Then*

$$|\bigcup_{\pi \in X} occ(v, \pi)| = \sum_{\pi \in X} |\bigcup occ(v, \pi)|.$$

Proof. Given Definition 4.12, we have $occ(v, \pi) \cap occ(v, \sigma) = \emptyset$, for any paths π, σ with $\pi \neq \sigma$. From this, the theorem follows easily. □

Next, we provide our definition of an occurrence of an edge $e \in E$ in a path:

Definition 4.13 (Occurrences of an edge)**.** *The* set $occ(e, \pi)$ of occurrences of an edge $e \in E$ in a path π *is defined as*

$$occ(e, \pi) = \{(\rho, \sigma) \mid \rho \circ e \circ \sigma = \pi\}.$$

Example 4.10. Reconsider path $\pi = v_{start}v_1v_3v_3v_3v_{end}$ from Example 4.9. We have

$$occ((v_3, v_3), \pi) = \{(v_{start}v_1, v_3v_{end}), (v_{start}v_1v_3, v_{end})\}).$$

As was the case with Definition 4.12, we have a useful equality:

Theorem 4.7. *Let $e \in E$ be an edge, and let X be a set of paths. Then*

$$|\bigcup_{\pi \in X} occ(e, \pi)| = \sum_{\pi \in X} |\bigcup occ(e, \pi)|.$$

Proof. Given Definition 4.13, we have $occ(e, \pi) \cap occ(e, \sigma) = \emptyset$, for any paths π, σ with $\pi \neq \sigma$. From this, the theorem follows easily. □

Now we define the slightly more elaborate notion of a covered occurrence of a node $v \in V$ in a path. It is conceptually similar to the notion of an occurrence of a node in a path, but only considers occurrences of v that occur inside a path of a given clip $\mathcal{S}$. Recall that by the *inside* of a path, we mean anything *between* its first and its ast node, but we do not include these border nodes themselves:

Definition 4.14 (Covered occurrence of a node)**.** *Let $\mathcal{S}$ be a clip. The set $occ(v, \pi, \mathcal{S})$* of $\mathcal{S}$-covered occurrences of a node $v \in V$ in a path π *is defined as*

$$occ(v, \pi, \mathcal{S}) = \{(\rho_1, \rho_2, \sigma_1, \sigma_2) \mid \rho_1 \circ \rho_2 \circ v \circ \sigma_1 \circ \sigma_2 = \pi, \rho_2 \neq \epsilon, \sigma_1 \neq \epsilon, \rho_2 \circ v \circ \sigma_1 \in paths(\mathcal{S})\}.$$

Example 4.11. Reconsider the contexts

$$\mathcal{C}_{v_3,1.1} = [\![\{(v_{start}, v_1)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,1.2} = [\![\{(v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!]$$

from Example 4.8. Moreover, reconsider path $\pi = v_{start}v_1v_3v_3v_3v_{end}$ from Example 4.10. We have

$$occ(v_3, \pi, \mathcal{C}_{v_3,1.1}) = \{(\epsilon, v_{start}v_1v_2, v_3, v_3v_{end})\};$$
$$occ(v_3, \pi, \mathcal{C}_{v_3,1.2}) = \{(v_{start}v_1v_2, v_3, v_3, v_{end}), (v_{start}v_1v_2v_3, v_3, v_{end}, \epsilon)\}.$$

As was the case with Definitions 4.12 and 4.13, we have a useful equality:

Theorem 4.8. *Let $v \in V$ be a node, let X be a set of paths, and let $\mathcal{S}$ be a clip. Then*

$$|\bigcup_{\pi \in X} occ(v, \pi, \mathcal{S})| = \sum_{\pi \in X} |\bigcup occ(v, \pi, \mathcal{S})|.$$

Proof. Given Definition 4.14, we have $occ(v, \pi, \mathcal{S}) \cap occ(v, \sigma, \mathcal{S}) = \emptyset$, for any paths π, σ with $\pi \neq \sigma$. From this, the theorem follows easily. □

The next step is to relate the occurrence of a given node $v \in V$ to the occurrence of its contexts, using linear constraints that we will later translate to IPET constraints:

Theorem 4.9 (Relating nodes to context)**.** *Let $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be pairwise divergent contexts of some node $v \in V$, such that $\bigcup_{1 \leq i \leq n} paths(\mathcal{C}_i)$ covers v. Then the following constraint holds:*

$$|occ(v, \pi)| = \sum_{1 \leq i \leq n} |occ(v, \pi, \mathcal{C}_i)|, \text{ for all } \pi \in \mathcal{U}.$$

Proof. We separately show the $\leq$ and the $\geq$ part of the equality:

1. Since the contexts $\mathcal{C}_i$ are pairwise divergent, we have

   $$\sum_{1 \leq i \leq n} |occ(v, \pi, \mathcal{C}_i)| = |\bigcup_{1 \leq i \leq n} occ(v, \pi, \mathcal{C}_i)|, \text{ for all } \pi \in \mathcal{U}, \text{ for all } \pi \in \mathcal{U}.$$

   Now, consider any path $\pi \in \mathcal{U}$ and any paths ρ, σ, with $\rho \circ v \circ \sigma = \pi$. Since $\bigcup_{1 \leq i \leq n} paths(\mathcal{C}_i)$ covers node v, there is some index i with $1 \leq i \leq n$, subpaths ρ_1, σ_2, and non-empty subpaths ρ_2, σ_1 with $\rho_1 \circ \rho_2 = \rho$ and $\sigma_1 \circ \sigma_2 = \sigma$, such that $\rho_2 \circ v \circ \sigma_1 \in paths(\mathcal{C}_i)$.

   Also, consider any paths ρ', σ', with $\rho' \circ v \circ \sigma' = \pi$ and $(\rho, \sigma) \neq (\rho', \sigma')$. Again, since $\bigcup_{1 \leq i \leq n} paths(\mathcal{C}_i)$ covers node v, there is some index i' with $1 \leq i' \leq n$, subpaths ρ'_1, σ'_2, and non-empty subpaths ρ'_2, σ'_1 with $\rho'_1 \circ \rho'_2 = \rho'$ and $\sigma'_1 \circ \sigma'_2 = \sigma'$, such that $\rho'_2 \circ v \circ \sigma'_1 \in paths(\mathcal{C}'_i)$. Since $(\rho, \sigma) \neq (\rho', \sigma')$, we also have $(\rho'_1, \rho'_2, \sigma'_1, \sigma'_2) \neq (\rho_1, \rho_2, \sigma_1, \sigma_2)$.

   We see that, for any two different elements (ρ, σ) and (ρ', σ') in $occ(v, \pi)$, we get different elements $(\rho_1, \rho_2, \sigma_1, \sigma_2)$ and $(\rho'_1, \rho'_2, \sigma'_1, \sigma'_2)$ in $\bigcup_{1 \leq i \leq n} occ(v, \pi, \mathcal{C}_i)$, hence

   $$|occ(v, \pi)| \leq |\bigcup_{1 \leq i \leq n} occ(v, \pi, \mathcal{C}_i)| = \sum_{1 \leq i \leq n} |occ(v, \pi, \mathcal{C}_i)|, \text{ for all } \pi \in \mathcal{U}.$$

2. Since the contexts $\mathcal{C}_i$ are pairwise divergent, we have

   $$\sum_{1 \leq i \leq n} |occ(v, \pi, \mathcal{C}_i)| = |\bigcup_{1 \leq i \leq n} occ(v, \pi, \mathcal{C}_i)|, \text{ for all } \pi \in \mathcal{U}, \text{ for all } \pi \in \mathcal{U}.$$

   Now, consider any path $\pi \in \mathcal{U}$. Choose any paths ρ_1, σ_2 and any paths $\rho_2 \neq \epsilon$, $\sigma_1 \neq \epsilon$, such that $\rho_1 \circ \rho_2 = \rho$, such that $\sigma_1 \circ \sigma_2 = \sigma$, and such that $\rho_2 \circ v \circ \sigma_1 \in paths(\mathcal{C}_i)$ with $1 \leq i \leq n$. Moreover, choose paths ρ'_1, σ'_2 and any paths $\rho'_2 \neq \epsilon$, $\sigma'_1 \neq \epsilon$, such that $\rho'_1 \circ \rho'_2 = \rho'$, such that $\sigma'_1 \circ \sigma'_2 = \sigma'$, and such that $\rho'_2 \circ v \circ \sigma'_1 \in paths(\mathcal{C}'_i)$ with $1 \leq i' \leq n$, and such that $(\rho_2, \sigma_1) \neq (\rho'_2, \sigma'_1)$. There are two cases:

   Case 1: $i = j$, i.e., paths $\rho_2 \circ v \circ \sigma_1$ and $\rho'_2 \circ v \circ \sigma'_1$ are in the same context. Then they are divergent, by Theorem 4.2.

   Case 2: $i \neq j$, i.e., paths $\rho_2 \circ v \circ \sigma_1$ and $\rho'_2 \circ v \circ \sigma'_1$ are in different contexts. Then they are divergent, by the original assumption that all contexts $\mathcal{C}_i$ are pairwise divergent.

   In both cases, paths $\rho_2 \circ v \circ \sigma_1$ and $\rho'_2 \circ v \circ \sigma'_1$ are divergent. Therefore, we have $(\sigma_1 \circ \sigma_2, \rho_1 \circ \rho_2) \neq (\sigma'_1 \circ \sigma'_2, \rho'_1 \circ \rho'_2)$. We see that, for any two different elements

(ρ_2, σ_1) and (ρ_2', σ_1') in $\bigcup_{1 \leq i \leq n} occ(v, \pi, \mathcal{C}_i)$, we get different elements $(\rho_1 \circ \rho_2, \sigma_1 \circ \sigma_2)$ and $(\rho_1' \circ \rho_2', \sigma_1' \circ \sigma_2')$ in $occ(v, \pi)$, hence

$$|occ(v, \pi)| \geq |\bigcup_{1 \leq i \leq n} occ(v, \pi, \mathcal{C}_i)| = \sum_{1 \leq i \leq n} |occ(v, \pi, \mathcal{C}_i)|, \text{ for all } \pi \in \mathcal{U}.$$

□

As a last step, we relate the occurrences of a given context $\mathcal{C} = [\![A, B]\!]$ to the occurrence of its entry edge A and its exit edges B, using, again, linear constraints that we will later translate to IPET constraints:

Theorem 4.10 (Relating contexts to entries and exits)**.** *Let $\mathcal{C} = [\![A, B]\!]$ be a context of some node $v \in V \setminus \{v_{start}, v_{end}\}$ that is neither the start node, nor the end node.*

Let X be the set of all edges $(x, z) \in E$, such that there exists an edge $(a, w) \in A$ with a path from node w to node x that contains only edges in $E \setminus (A \cup B)$, such that there exists an edge $(u, v) \in E \setminus (A \cup B)$ with a path from node x to node u that contains only edges in $E \setminus (A \cup B)$, and such there exists no path from node x to node v that contains node z and only edges in $E \setminus (A \cup B)$.

Let Y be the set of all edges $(z, y) \in E$, such that there exists an edge $(w, b) \in B$ with a path from node y to node w that contains only edges in $E \setminus (A \cup B)$, such that there exists an edge $(v, u) \in E \setminus (A \cup B)$ with a path from node u to node y that contains only edges in $E \setminus (A \cup B)$, and such there exists no path from node v to node y that contains node z and only edges in $E \setminus (A \cup B)$.

Then the following constraints hold for every path $\pi \in paths(\mathcal{C})$:

$$|occ(v, \pi, \mathcal{C})| \leq \sum_{a \in A} |occ(a, \pi)| - \sum_{x \in X} |occ(x, \pi)|;$$
$$|occ(v, \pi, \mathcal{C})| \leq \sum_{b \in B} |occ(b, \pi)| - \sum_{y \in Y} |occ(y, \pi)|.$$

Proof. We give a proof for the first inequality only. The proof for the second inequality is completely symmetric.

By the definition of *occ* and by the observation that

$$occ(w, \pi) \cap occ(w', \pi) = \emptyset, \text{ for all } w, w' \in V, w \neq w', \pi \in \mathcal{U},$$

we have

$$\sum_{a \in A} |occ(a, \pi)| = |\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, a \in A\}|;$$
$$\sum_{x \in X} |occ(x, \pi)| = |\{(\rho', \sigma') \mid \rho' \circ x \circ \sigma' = \pi, x \in X\}|.$$

If we consider the definition of X, we see that ρ' must contain an occurrence of some edge $a \in A$, such that there is no subsequent occurrence of any edge $y \in A \cup B$ in ρ',

i.e., we have

$$\{(\rho', \sigma') \mid \rho' \circ x \circ \sigma' = \pi, x \in X\} = \{(\rho', \sigma') \mid \rho' \circ x \circ \sigma' = \pi, \rho \circ a \circ \tau = \rho', a \in A, \\ x \in X, \nexists(y \in A \cup B, \tau_1, \tau_2) : \tau_1 \circ y \circ \tau_2 = \tau\}.$$

We are merely interested in the number of elements in the latter set, not in the elements themselves. This allows us to make use of the following equality:

$$\begin{aligned} &|\{(\rho', \sigma') \mid \rho' \circ x \circ \sigma' = \pi, \rho \circ a \circ \tau = \rho', a \in A, \\ &\quad x \in X, \nexists(y \in A \cup B, \tau_1, \tau_2) : \tau_1 \circ y \circ \tau_2 = \tau\}| \\ &= |\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, \tau \circ x \circ \sigma' = \sigma, a \in A, \\ &\quad x \in X, \nexists(y \in A \cup B, \tau_1, \tau_2) : \tau_1 \circ y \circ \tau_2 = \tau\}|. \end{aligned}$$

Since

$$\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, \tau \circ x \circ \sigma' = \sigma, a \in A, \\ x \in X, \nexists(y \in A \cup B, \tau_1, \tau_2) : \tau_1 \circ y \circ \tau_2 = \tau\}$$

is a subset of

$$\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, a \in A\},$$

we have

$$\sum_{a \in A} |occ(a, \pi)| - \sum_{x \in X} |occ(x, \pi)| = |\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, a \in A\} \setminus \\ \{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, \tau \circ x \circ \sigma' = \sigma, a \in A, \\ x \in X, \nexists(y \in A \cup B, \tau_1, \tau_2) : \tau_1 \circ y \circ \tau_2 = \tau\}|$$

Expansion of the set subtraction yields

$$\sum_{a \in A} |occ(a, \pi)| - \sum_{x \in X} |occ(x, \pi)| = |\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, a \in A, \\ \nexists(x \in X, y \in A \cup B, \tau_1, \tau_2, \sigma') : \\ \tau_1 \circ y \circ \tau_2 \circ x \circ \sigma' = \sigma\}|.$$

Now, choose any element from $occ(v, \pi, [\![A, B]\!])$, i.e., choose any paths ρ_1, σ_2, and any paths $\rho_2 \neq \epsilon$, $\sigma_1 \neq \epsilon$, with $\rho_1 \circ \rho_2 \circ v \circ \sigma_1 \circ \sigma_2 = \pi$, and where $\rho_2 \circ v \circ \sigma_1 \in paths([\![A, B]\!])$. By the definition of a context, path ρ_2 must start with an entry edge $a \in A$, must end with an exit edge $b \in B$, and cannot contain any further occurrences of any edge in $A \cup B$. Moreover, path $\rho_2 \circ v \circ \sigma_1$ cannot contain any edge $x \in X$. Therefore, $(\rho_1 \circ \rho_2, \sigma_1 \circ \sigma_2)$ is an element of

$$\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, a \in A, \\ \nexists(x \in X, y \in A \cup B, \tau_1, \tau_2, \sigma') : \tau_1 \circ y \circ \tau_2 \circ x \circ \sigma' = \sigma\}.$$

Next, choose another element from $occ(v, \pi, [\![A, B]\!])$, i.e., choose paths ρ_1', σ_2', and any paths $\rho_2' \neq \epsilon$, $\sigma_1' \neq \epsilon$, with $\rho_1' \circ \rho_2' \circ v \circ \sigma_1' \circ \sigma_2' = \pi$, and where $\rho_2' \circ v \circ \sigma_1' \in paths([\![A, B]\!])$, with $(\rho_1', \rho_2', \sigma_1', \sigma_2') \neq (\rho_1, \rho_2, \sigma_1, \sigma_2)$. By the definition of a context, path ρ_2' must start

with an entry edge $a \in A$, must end with an exit edge $b \in B$, and cannot contain any further occurrences of any edge in $A \cup B$. Moreover, path $\rho_2' \circ v \circ \sigma_1'$ cannot contain any edge $x \in X$. Therefore, $(\rho_1' \circ \rho_2', \sigma_1' \circ \sigma_2')$ is an element of

$$\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, a \in A, \\ \nexists (x \in X, y \in A \cup B, \tau_1, \tau_2, \sigma') : \tau_1 \circ y \circ \tau_2 \circ x \circ \sigma' = \sigma\}.$$

By Theorem 4.2, the paths $\rho_2' \circ v \circ \sigma_1'$ and $\rho_2 \circ v \circ \sigma_1$ are divergent, and therefore we have $(\rho_1 \circ \rho_2, \sigma_1 \circ \sigma_2) \neq (\rho_1' \circ \rho_2', \sigma_1' \circ \sigma_2')$. We see that, for any two different elements $(\rho_1, \rho_2, \sigma_1, \sigma_2)$ and $(\rho_1', \rho_2', \sigma_1', \sigma_2')$ from $occ(v, \pi, \mathcal{C})$, we get different elements $(\rho_1 \circ \rho_2, \sigma_1 \circ \sigma_2)$ and $(\rho_1' \circ \rho_2', \sigma_1' \circ \sigma_2')$ in

$$\{(\rho, \sigma) \mid \rho \circ a \circ \sigma = \pi, a \in A, \\ \nexists (x \in X, y \in A \cup B, \tau_1, \tau_2, \sigma') : \tau_1 \circ y \circ \tau_2 \circ x \circ \sigma' = \sigma\},$$

hence

$$|occ(v, \pi, \mathcal{C})| \leq \sum_{a \in A} |occ(a, \pi)| - \sum_{x \in X} |occ(x, \pi)|.$$

□

Example 4.12. Reconsider contexts

$$\mathcal{C}_{v_3,1.0} = [\![\{(v_{start}, v_1), (v_3, v_3)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,2.0} = [\![\{(v_1, v_2)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,1.1} = [\![\{(v_{start}, v_1)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,1.2} = [\![\{(v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!].$$

of node v_3 from Examples 4.7 and 4.8. First, consider context $\mathcal{C}_{v_3,1.0}$. We have

$$X = \{(v_1, v_2)\} \quad \text{and} \quad Y = \emptyset,$$

which yields the following constraints:

$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.0})| \leq |occ((v_{start}, v_1), \pi)| + |occ((v_3, v_3), \pi)| - |occ((v_1, v_2), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.0})| \leq |occ((v_3, v_3), \pi)| + |occ((v_3, v_{end}), \pi)|.$$

For context $\mathcal{C}_{v_3,2.0}$, we have

$$X = \emptyset \quad \text{and} \quad Y = \emptyset,$$

which yields the following constraints:

$$|occ(v_3, \pi, \mathcal{C}_{v_3,2.0})| \leq |occ((v_1, v_2), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,2.0})| \leq |occ((v_3, v_3)), \pi)| + |occ((v_3, v_{end}), \pi)|.$$

For context $\mathcal{C}_{v_3,1.1}$, we have

$$X = \{(v_1, v_2)\} \quad \text{and} \quad Y = \emptyset,$$

which yields the following constraints:

$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.1})| \leq |occ((v_{start}, v_1), \pi)| - |occ((v_1, v_2), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.1})| \leq |occ((v_3, v_3), \pi)| + |occ((v_3, v_{end}), \pi)|.$$

For context $\mathcal{C}_{v_3,1.2}$, we have

$$X = \emptyset \quad \text{and} \quad Y = \emptyset,$$

which yields the following constraints:

$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.2})| \leq |occ((v_3, v_3), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.2})| \leq |occ((v_3, v_3), \pi)| + |occ((v_3, v_{end}), \pi)|.$$

4.7 Timed Traces and Clips

In this section we describe how the observed execution times of a given node $v \in V$—obtained from measurements—can be assigned to individual contexts of v.

We assume that measurements are provided as timed traces. An individual timed trace indicates the execution sequence of nodes during a particular run of the software on the target platform, as well as the observed execution duration for each occurrence of each node in the sequence. Formally, we define a timed trace as follows:

Definition 4.15 (Timed trace)**.** *A* timed trace of a program $\mathcal{P}$ *is a finite sequence* $\pi = (v_1, t_1) \dots (v_n, t_n)$*, where* $v_1 \dots v_n$ *is a path in the program's CFG* $\mathcal{G}_{\mathcal{P}} = (V, E, v_{start}, v_{end})$*, and where* $t_1, \dots, t_n$ *are the associated observed execution times of* $v_1, \dots, v_n$*.*

The maximal observed execution time (MOET) of a given node $v \in V$ within a timed trace π is the maximal execution time that is associated with any occurrence of v *inside* π. Recall that by the *inside* of a path, we mean anything *between* the first and the last node, but we do not include the border nodes themselves. Formally, we define:

Definition 4.16 (MOET of node in timed trace)**.** *The maximal observed execution time (MOET)* $moet_{v,\pi}$ *of a node* $v \in V$ *inside a timed trace* π *is defined as the maximum over all associated execution times of* v *occurring inside* π*, i.e.,*

$$moet_{v,\pi} = max\{t_i \mid \pi = (v_1, t_1) \dots (v_n, t_n), v_i = v, 1 < i < n\}.$$

Note that $moet_{v,\pi}$ is undefined, if π does not contain any occurrence of v.

Example 4.13. Consider the timed traces

$\pi_1 = (v_{start}, 0)(v_1, 40)(v_3, 20)(v_{end}, 0);$ $\pi_2 = (v_3, 5)(v_3, 4)(v_3, 4)(v_3, 4);$
$\pi_3 = (v_{start}, 0)(v_1, 40)(v_3, 25)(v_{end}, 0);$ $\pi_4 = (v_{start}, 0)(v_1, 40)(v_2, 20);$
$\pi_5 = (v_{start}, 0)(v_1, 40)(v_3, 30)(v_3, 20)(v_{end}, 0);$ $\pi_6 = (v_3, 5);$
$\pi_7 = (v_{start}, 0)(v_1, 45)(v_2, 15)(v_3, 10)(v_{end}, 0).$

The MOETs of node v_3 are as follows:

$$moet_{v_3,\pi_1} = 20; \qquad moet_{v_3,\pi_2} = 4;$$
$$moet_{v_3,\pi_3} = 25; \qquad moet_{v_3,\pi_4} \text{ undefined};$$
$$moet_{v_3,\pi_5} = 30; \qquad moet_{v_3,\pi_6} \text{ undefined};$$
$$moet_{v_3,\pi_7} = 10.$$

The notion of a MOET of a node in a path can easily be lifted to sets of timed traces:

Definition 4.17 (MOET of node in set of traces)**.** *The* maximal observed execution time (MOET) $moet_{v,\mathcal{T}}$ of a node $v \in V$ over a set of timed traces $\mathcal{T}$ *is defined as the maximum of all maximal observed execution times of* $v \in V$ *in any timed trace* $\pi \in \mathcal{T}$*, i.e.,*

$$moet_{v,\mathcal{T}} = \max\{moet_{v,\pi} \mid \pi \in \mathcal{T}\}.$$

Note that $moet_{v,\mathcal{T}}$ is undefined, if none of the timed traces in $\mathcal{T}$ contains an occurrence of v.

Example 4.14. Reconsider the timed traces $\pi_1, \ldots, n_7$ from Example 4.13. The MOET of node v_3 over the set $\mathcal{T} = \{\pi_1, \ldots, \pi_7\}$ of timed traces is

$$moet_{v_3,\mathcal{T}} = 30.$$

Definition 4.18 (Untimed trace)**.** *The corresponding* untimed trace $\overline{\pi}$ of a timed trace $\pi = (v_1, t_1) \ldots (v_n, t_n)$ *is the sequence of nodes occurring in* π*, i.e.,*

$$\overline{\pi} = v_1 \ldots v_n.$$

Example 4.15. Reconsider the times traces $\pi_1, \ldots, \pi_7$ from Example 4.13. The corresponding untimed traces are

$$\overline{\pi}_1 = v_{start} v_1 v_3 v_{end}; \qquad \overline{\pi}_2 = v_3 v_3 v_3 v_3;$$
$$\overline{\pi}_3 = v_{start} v_1 v_3 v_{end}; \qquad \overline{\pi}_4 = v_{start} v_1 v_2;$$
$$\overline{\pi}_5 = v_{start} v_1 v_3 v_3 v_{end}; \qquad \overline{\pi}_6 = v_3;$$
$$\overline{\pi}_7 = v_{start} v_1 v_2 v_3 v_{end}.$$

Definition 4.19 (MOET of node in clip)**.** *The* maximal observed execution time (MOET) $moet_{v,\mathcal{S},\mathcal{T}}$ of a node $v \in V$ in clip $\mathcal{S}$ over a set of timed traces $\mathcal{T}$ *is the MOET of* v *over the set of all timed subtraces in* $\mathcal{T}$ *with corresponding untimed traces that are paths in* $\mathcal{S}$*, i.e.,*

$$moet_{v,\mathcal{S},\mathcal{T}} = \max\{moet_{v,\pi} \mid \overline{\pi} \in paths(\mathcal{S}), \sigma \circ \pi \circ \rho \in \mathcal{T}\}.$$

Note that $moet_{v,\mathcal{S},\mathcal{T}}$ is undefined, if none of the timed traces in $\mathcal{T}$ contains an occurrence of v, or if none of the timed traces in $\mathcal{T}$ that contain an occurrence of v has a corresponding untimed trace that is a path in clip $\mathcal{S}$.

Example 4.16. Recall that a context $\mathcal{C}$ of a node v is just a clip with an additional constraint on the number of times that v is allowed to occur within the paths of $\mathcal{C}$. So

reconsider contexts

$$\mathcal{C}_{v_3,1.0} = [\![\{(v_{start}, v_1), (v_3, v_3)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,2.0} = [\![\{(v_1, v_2)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,1.1} = [\![\{(v_{start}, v_1)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,1.2} = [\![\{(v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!]$$

of node v_3 from Examples 4.7 and 4.8. Also reconsider the timed traces $\pi_1, \ldots, \pi_7$ from Example 4.13. The MOETs of node v_3 within these contexts over the set $\mathcal{T} = \{\pi_1, \ldots, \pi_7\}$ of timed traces are

$$moet_{v_3,\mathcal{C}_{v_3,1.0},\mathcal{T}} = 30;$$
$$moet_{v_3,\mathcal{C}_{v_3,2.0},\mathcal{T}} = 10;$$
$$moet_{v_3,\mathcal{C}_{v_3,1.1},\mathcal{T}} = 30;$$
$$moet_{v_3,\mathcal{C}_{v_3,1.2},\mathcal{T}} = 4.$$

Theorem 4.11 (MOET reduction)**.** *Let $\mathcal{S}$ be a clip, and let $\mathcal{T}$ be a set of timed traces. Then*

$$moet_{v,\mathcal{S},\mathcal{T}} \leq moet_{v,\mathcal{T}}.$$

Proof. It is easy to see that $\{moet_{v,\pi} \mid \overline{\pi} \in paths(\mathcal{S}), \sigma \circ \pi \circ \rho \in \mathcal{T}\} \subseteq \{moet_{v,\pi} \mid \pi \in \mathcal{T}\}$. Hence the property follows immediately. □

4.8 Finding Contexts for MBTA

In this section, we describe an algorithm for obtaining, for any given node $v \in V \setminus \{v_{start}, v_{end}\}$, a set $\{\mathcal{C}_{v,1}, \ldots \mathcal{C}_{v,n(v)}\}$ of contexts of v, with pairwise divergent sets of paths that together cover v. Moreover, the contexts are constructed in such a way that they are associated with different maximal observed execution times.

To construct a suitable set of contexts for some node $v \in V \setminus \{v_{start}, v_{end}\}$, the algorithm tests the MOET $moet_{v,\mathcal{C},\mathcal{T}}$ of v in various candidate contexts $\mathcal{C}$, over the provided set $\mathcal{T}$ of timed traces.

However, we have noted before that $moet_{v,\mathcal{C},\mathcal{T}}$ needs not be defined under all circumstances: More precisely, $moet_{v,\mathcal{C},\mathcal{T}}$ is undefined, if none of the timed traces in $\mathcal{T}$ contains an occurrence of v, or if none of the timed traces in $\mathcal{T}$ that contain an occurrence of v has a corresponding untimed trace that is a path in context $\mathcal{C}$.

In MBTA the set $\mathcal{T}$ of timed traces is obtained by performing measurements. In that case, $moet_{v,\mathcal{C},\mathcal{T}}$ is undefined, if node v was not reached by any measurement, or if none of the paths in $\mathcal{C}$ was exhibited by any measurement.

There are two basic strategies for handling such cases of missing measurements:

Conservative approach: In this approach, the algorithm by default attributes missing measurements to insufficient coverage of the temporal behavior. It assumes that suitable timed traces can, in principle, be found, and conservatively substitutes the global MOET $moet_{v,\mathcal{T}}$ for $moet_{v,\mathcal{S},\mathcal{T}}$.

Progressive approach: In this approach, the algorithm by default attributes missing measurements to infeasible paths. It assumes that suitable timed traces can, in principle, not be found, substitutes 0 for $moet_{v,\mathcal{S},\mathcal{T}}$, and stipulates that the corresponding clip is infeasible.

To simplify the presentation of the algorithm, we assume that $moet_{v,\mathcal{S},\mathcal{T}}$ is always defined, i.e., there is always at least one matching measurement. The handling of missing measurements does not provide additional insight into the fundamental idea of the algorithm.

Algorithm 4.1 Find a set of contexts.

Require: CFG $\mathcal{G}_\mathcal{P} = (V, E, v_{start}, v_{end})$, node $v \in V \setminus \{v_{start}, v_{end}\}$, timed traces $\mathcal{T}$

find cut edges

$Q \leftarrow \{(v_{start}, q) \in E\}$ — all outgoing edges of node v_{start}
$B \leftarrow \{(v, b) \in E\}$ — all outgoing edges of node v
$R \leftarrow \{(w, r) \in E \mid (r, v) \in E^*\}$ — all edges that reach node v
$A \leftarrow (Q \cup B) \cap R$ — $[\![A, B]\!]$ is simple-history context of node v
$X \leftarrow \emptyset$
for $(u, w) \in E$ **do**
  $O_u \leftarrow \{(u, x) \in E\}$ — all outgoing edges of node u
  if $moet_{v,[\![\{(u,w)\},B]\!],\mathcal{T}} \leq moet_{v,[\![O_u,B]\!],\mathcal{T}}$ **then**
    $X \leftarrow X \cup \{(u, w)\}$ — collect cut edge
  end if
end for

vertical context split

$Y \leftarrow \{(u, w) \in E \mid (x, y) \in A, (y, u) \in (E \setminus (A \cup B \cup X))^*\}$ — restr. reachable from A
$Z \leftarrow \{(u, w) \in E \mid (x, y) \in X, (y, u) \in (E \setminus (A \cup B \cup X))^*\}$ — restr. reachable from X
$A_1 \leftarrow A$, $B_1 \leftarrow (B \cup X) \cap Y$, $\mathcal{C}_{v,1} \leftarrow [\![A_1, B_1]\!]$ — first context from split
$A_2 \leftarrow X$, $B_2 \leftarrow (B \cup X) \cap Z$, $\mathcal{C}_{v,2} \leftarrow [\![A_2, B_2]\!]$ — second context from split

horizontal context split

$M \leftarrow \emptyset$
for $i \in \{1, 2\}$ **do**
  $\sim_i = \{(x, y) \mid moet_{v,[\![\{x\},B_i]\!],\mathcal{T}} = moet_{v,[\![\{y\},B_i]\!],\mathcal{T}}, x, y \in A_i\}$ — MOET-equivalence
  $\mathcal{D} \leftarrow A/\sim_i$ — factor entry edges by MOET-equivalence
  for $D \in \mathcal{D}$ **do**
    $Z_D \leftarrow \{(u, w) \mid (d, x) \in D, (x, u) \in (E \setminus (A_i \cup B_i))^*\}$ — restr. reachable from D
    $M \leftarrow M \cup \{[\![D, B_i \cap Z_D]\!]\}$ — collect context
  end for
end for
return M — a set of suitable contexts

The following is a rather informal description of how our algorithm works. A more precise formulation is given as Algorithm 4.1. Our algorithm proceeds as follows:

1. The algorithm initially finds the set Q all edges $(v_{start}, q) \in E$, the set B of all edges $(v, b) \in E$, and the set R of all edges $(w, r) \in E$, such that there is a path from r to v. Set R can easily be found by performing a backward depth-first

search, starting from node v. Sets Q and B can be found from an adjacency list or adjacency matrix of the CFG.

2. Let $A = (Q \cup B) \cap R$. Note that $\mathcal{C} = [\![A, B]\!]$ is the simple-history context of v.

3. For each edge $(u, w) \in E$, the algorithm checks the condition

$$moet_{v,[\![\{(u,w)\},B]\!],\mathcal{T}} \leq moet_{v,[\![O_u,B]\!],\mathcal{T}},$$

where $O_u = \{(u, x) \in E\}$ is the set of all outgoing edges of node u. The condition is a test if context $[\![\{(u, w)\}, B]\!]$ of v—which has edge (u, w) as its only entry edge—provides a lower MOET for node v than context $(O_u, B]\!]$ of v—which has *all* edges starting from node u as entry edges. If this is true, then the context $[\![\{(u, w)\}, B]\!]$ captures a potential special case of executing v with a reduced execution time. Let X be the set of all edges $(u, w) \in E$ for which the condition holds. It is therefore reasonable to consider $[\![X, B]\!]$ as a separate context of v.

4. The next step of the algorithm is a vertical context split: The algorithm finds the set Y of all edges $(u, w) \in E$ such that there exists a path from some edge in A to node u that contains only edges in $E \setminus (A \cup B \cup X)$. It also finds the set Z of all edges (u, w) such that there exists a path from some edge in X to node u that contains only edges in $E \setminus (A \cup B \cup X)$. Let $A_1 = A$, $B_1 = (B \cup X) \cap Y$, $A_2 = X$, and $B_2 = (B \cup X) \cap Z$. Note that $\mathcal{C}_{v,1} = [\![A_1, B_1]\!]$ and $\mathcal{C}_{v,2} = [\![A_2, B_2]\!]$ are contexts with $paths(\mathcal{C}_{v,1}) \cap paths(\mathcal{C}_{v,2}) = \emptyset$ of v that cover node v.

5. The final step of the algorithm is a horizontal split of context $\mathcal{C}_{v,i}$, for $i \in \{1, 2\}$: In this step the algorithm creates a partition $\mathcal{D}_i$ of set A_i by the MOET of node v, i.e., $\mathcal{D}_i = A_i/\sim_i$, where $\sim_i$ is the following equivalence relation:

$$x \sim_i y \quad \text{iff} \quad moet_{v,[\![\{x\},B_i]\!],\mathcal{T}} = moet_{v,[\![\{y\},B_i]\!],\mathcal{T}} \quad \text{for all } x, y \in A_i.$$

For each set $D \in \mathcal{D}_i$, the algorithm finds the set Z_D of all edges $(u, w) \in E$, such that there exists a path from some edge in D to node u that contains only edges in $E \setminus (A_i \cup B_i)$.

6. The set of suitable contexts produced by the algorithm is

$$M = \{[\![D, B_1 \cap Z_D]\!] \mid D \in \mathcal{D}_1\} \cup \{[\![D, B_2 \cap Z_D]\!] \mid D \in \mathcal{D}_2\}.$$

Example 4.17. Reconsider CFG $\mathcal{G}$ from Example 3.1. Also, reconsider the timed traces

$\pi_1 = (v_{start}, 0)(v_1, 40)(v_3, 20)(v_{end}, 0)$; $\pi_2 = (v_3, 5)(v_3, 4)(v_3, 4)(v_3, 4)$;
$\pi_3 = (v_{start}, 0)(v_1, 40)(v_3, 25)(v_{end}, 0)$; $\pi_4 = (v_{start}, 0)(v_1, 40)(v_2, 20)$;
$\pi_5 = (v_{start}, 0)(v_1, 40)(v_3, 30)(v_3, 20)(v_{end}, 0)$; $\pi_6 = (v_3, 5)$;
$\pi_7 = (v_{start}, 0)(v_1, 45)(v_2, 15)(v_3, 10)(v_{end}, 0)$

from Example 4.13. We apply Algorithm 4.1 on CFG $\mathcal{G}$, node v_3, and the set of timed traces $\mathcal{T} = \{\pi_1, \ldots, \pi_7\}$:

1. The algorithm initially finds

$$Q = \{(v_{start}, v_1)\};$$
$$B = \{(v_3, v_3), (v_3, v_{end})\};$$
$$R = \{(v_{start}, v_1), (v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_3)\}.$$

2. The algorithm sets

$$A = (Q \cup B) \cap R = \{(v_{start}, v_1), (v_3, v_3)\},$$

   The clip $\mathcal{C}_{v_3} = [\![A, B]\!]$ is indeed the simple-history context of v_3.

3. The algorithm finds

$$moet_{v_3,[\![\{(v_1,v_2)\},B]\!]} = 10 \leq moet_{v_3,[\![\{(v_1,v_2),(v_1,v_3)\},B]\!]} = 30;$$
$$X = \{(v_1, v_2)\}.$$

4. The Algorithm performs a vertical context split along X:

$$Y = \{(v_1, v_2), (v_1, v_3), (v_3, v_3), (v_3, v_{end})\};$$
$$Z = \{(v_2, v_3), (v_3, v_3), (v_3, v_{end})\};$$
$$A_1 = A = \{(v_{start}, v_1), (v_3, v_3)\};$$
$$B_1 = (B \cup X) \cap Y = \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\};$$
$$A_2 = X = \{(v_1, v_2)\};$$
$$B_2 = (B \cup X) \cap Z = \{(v_3, v_3), (v_3, v_{end})\}.$$
$$\mathcal{C}_{v,1} = [\![A_1, B_1]\!]$$
$$\mathcal{C}_{v,2} = [\![A_2, B_2]\!]$$

5. The algorithm finds

$$moet_{v_3,[\![\{(v_{start},v_1)\},B_1]\!]} = 30 \neq moet_{v_3,[\![\{(v_3,v_3)\},B_2]\!]} = 4;$$

$$\mathcal{D}_1 = \{\{(v_{start}, v_1)\}, \{(v_3, v_3)\}\};$$
$$\mathcal{D}_2 = \{\{(v_1, v_2)\}\};$$
$$Z_{\{(v_{start},v_1)\}} = \{(v_1, v_2), (v_1, v_3), (v_3, v_3), (v_3, v_{end})\};$$
$$Z_{\{(v_3,v_3)\}} = \{(v_3, v_3), (v_3, v_{end})\};$$
$$Z_{\{(v_1,v_2)\}} = \{(v_2, v_3), (v_3, v_3), (v_3, v_{end})\};$$

6. The algorithm produces the set of contexts

$$\begin{aligned} M = \{&[\![\{(v_{start}, v_1)\}, B_1 \cap Z_{\{(v_{start},v_1)\}}]\!], \\ &[\![\{(v_3, v_3)\}, B_1 \cap Z_{\{(v_3,v_3)\}}]\!], \\ &[\![\{(v_1, v_2)\}, B_2 \cap Z_{\{(v_1,v_2)\}}]\!]\} \\ = \{&[\![\{(v_{start}, v_1)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!], \\ &[\![\{(v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!], \\ &[\![\{(v_1, v_2)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!]\}. \end{aligned}$$

Theorem 4.12. *Given a CFG $\mathcal{G}_{\mathcal{P}} = (V, E, v_{start}, v_{end})$, a node $v \in V \setminus \{v_{start}, v_{end}\}$, and a set of timed traces $\mathcal{T}$, Algorithm 4.1 returns a set of contexts $M = \{\mathcal{C}_{v,1}, \ldots, \mathcal{C}_{v,n}\}$ of node v, such that $paths(\mathcal{C}_{v,i})$ and $paths(\mathcal{C}_{v,j})$ are divergent, for $1 \leq i \leq n$, $1 \leq j \leq n$, and $i \neq j$, and such that $\bigcup_{1 \leq i \leq n} paths(\mathcal{C}_{v,i})$ covers node v.*

Proof. The algorithm starts by constructing the simple-history context $[\![A, B]\!]$ of node v. By Theorem 4.3, $paths([\![A, B]\!])$ covers node v. Next, it obtains contexts $[\![A_1, B_1]\!]$ and $[\![A_1, B_1]\!]$ of node v, by performing a vertical split of context $[\![A, B]\!]$. By Theorem 4.4, $paths([\![A_1, B_1]\!]) \cup paths([\![A_2, B_2]\!])$ covers node v, and $[\![A_1, B_1]\!]$ and $[\![A_2, B_2]\!]$ are divergent. The algorithm then performs a horizontal split of contexts $[\![A_1, B_1]\!]$ and $[\![A_2, B_2]\!]$, thus obtaining contexts

$$\begin{aligned} M_1 &= \{[\![D, B_1 \cap Z_D]\!] \mid D \in \mathcal{D}_1\}, \text{ and} \\ M_2 &= \{[\![D, B_2 \cap Z_D]\!] \mid D \in \mathcal{D}_2\}. \end{aligned}$$

By Theorem 4.5,

$$paths([\![A_1, B_1]\!]) \cup \bigcup_{D \in \mathcal{D}_2} paths([\![D, B_2 \cap Z_D]\!])$$

covers node v. Again, by Theorem 4.5,

$$\begin{aligned} &\bigcup_{D \in \mathcal{D}_1} paths([\![D, B_1 \cap Z_D]\!]) \cup \bigcup_{D \in \mathcal{D}_2} paths([\![D, B_1 \cap Z_D]\!]) \\ = &\bigcup_{D \in \mathcal{D}_1 \cup \mathcal{D}_2} paths([\![D, B_1 \cap Z_D]\!]) \end{aligned}$$

covers node v.

By Theorem 4.5, $paths([\![D, B_1 \cap Z_D]\!])$ and $paths([\![D', B_1 \cap Z_{D'}]\!])$ are divergent, for $D, D' \in \mathcal{D}_1$, with $D \neq D'$. Also, $paths([\![D, B_2 \cap Z_D]\!])$ and $paths([\![D', B_2 \cap Z_{D'}]\!])$ are divergent, for $D, D' \in \mathcal{D}_2$, with $D \neq D'$.

Now choose any $D \in \mathcal{D}_1$ and $D' \in \mathcal{D}_2$. Since the contexts $[\![A_1, B_1]\!]$ and $[\![A_2, B_2]\!]$ are divergent, and since

$$\begin{aligned} paths([\![D, B_1 \cap Z_D]\!]) &\subseteq [\![A_1, B_1]\!], \text{ and} \\ paths([\![D', B_2 \cap Z_{D'}]\!]) &\subseteq [\![A_2, B_2]\!], \end{aligned}$$

it follows that $paths([\![D, B_1 \cap Z_D]\!])$ and $paths([\![D', B_2 \cap Z_{D'}]\!])$ are also divergent. □

4.9 Instantiating Context-Sensitive IPET

We are now able to put the results from Sections 4.5 through 4.8 together, to obtain an instantiation of context-sensitive IPET:

1. We use Algorithm 4.1 to generate a set $\mathcal{Q}_v = \{\mathcal{C}_{v,1}, \ldots \mathcal{C}_{v,n(v)}\}$ of suitable contexts, for every node $v \in V \setminus \{v_{start}, v_{end}\}$. By way of Theorem 4.12, $paths(\mathcal{C}_{v,i})$ and $paths(\mathcal{C}_{v,j})$ are divergent, for $1 \leq i \leq n(v)$, $1 \leq j \leq n(n)$, and $i \neq j$, and $\bigcup_{1 \leq i \leq n(v)} paths(\mathcal{C}_{v,i})$ covers node v. Therefore, Theorem 4.9 applies, hence Requirement 4.2 is met.

2. For each node $v \in V \setminus \{v_{start}, v_{end}\}$, we interpret the individual contexts $\mathcal{C}_{v,1}, \ldots \mathcal{C}_{v,n(v)}$ as individual execution scenarios $E_{v,1}, \ldots E_{v,n(v)}$.

3. We use the MOETs $moet_{v,\mathcal{C}_{v,1},\mathcal{T}}, \ldots, moet_{v,\mathcal{C}_{v,n(v)},\mathcal{T}}$ of each context as WCET estimates $\widetilde{wcet}_{v,1}, \ldots, \widetilde{wcet}_{v,n(v)}$.

4. We use the construction in Theorem 4.10 to infer ILP constraints over our execution scenario variables. The translation of the linear constraints presented in the theorem is straightforward: For example, the linear constraint

$$|occ(v, \pi, \mathcal{C})| \leq \sum_{e \in A} |occ(e, \pi)| - \sum_{e \in X} |occ(e, \pi)|, \text{ for every path } \pi \in paths(\mathcal{C})$$

   translates to a corresponding IPET constraint

$$f_{v,i} \leq \sum_{e \in A} f_e - \sum_{e \in X} f_e.$$

   By adding these constraints, we fulfill Requirement 4.3. Moreover, Requirement 4.1 is fulfilled as a consequence of Theorem 4.11.

Example 4.18. Reconsider CFG $\mathcal{G}$ from Example 3.1. Example 3.2 provides an IPET problem for $\mathcal{G}$, constructed for some hypothetical WCET estimates of the individual nodes. We can reuse the constraints from that IPET problem to construct a context-sensitive IPET problem for the timed trace $\mathcal{T} = \{\pi_1, \ldots, \pi_7\}$ from Example 3.1.

1. The latter example has already illustrated the application of Algorithm 4.1, to obtain a set

$$M = \{\mathcal{C}_{v_3,1.1}, \mathcal{C}_{v_3,1.2}, \mathcal{C}_{v_3,2.0}\}$$

   of suitable contexts for node v_3, where

$$\mathcal{C}_{v_3,1.1} = [\![\{(v_{start}, v_1)\}, \{(v_1, v_2), (v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,1.2} = [\![\{(v_3, v_3)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!];$$
$$\mathcal{C}_{v_3,2.0} = [\![\{(v_1, v_2)\}, \{(v_3, v_3), (v_3, v_{end})\}]\!].$$

2. We interpret the individual contexts $\mathcal{C}_{v_3,1.1}, \mathcal{C}_{v_3,1.2}, \mathcal{C}_{v_3,2.0}$ as individual execution scenarios $E_{v_3,1}, E_{v_3,2}, E_{v_3,3}$ with associated variables $f_{v_3,1}, f_{v_3,2}, f_{v_3,3}$.

3. We use the MOETs

$$moet_{v_3,\mathcal{C}_{v_3,1.1},\mathcal{T}} = 30;$$
$$moet_{v_3,\mathcal{C}_{v_3,1.2},\mathcal{T}} = 4;$$
$$moet_{v_3,\mathcal{C}_{v_3,2.0},\mathcal{T}} = 10.$$

of contexts $\mathcal{C}_{v_3,1.1}, \mathcal{C}_{v_3,1.2}, \mathcal{C}_{v_3,2.0}$—which we have calculated in Example 4.13—as WCET estimates $\widetilde{wcet}_{v_3,1}, \widetilde{wcet}_{v_3,2}, \widetilde{wcet}_{v_3,3}$.

4. We use the construction in Theorem 4.10 to infer the IPET constraints. We have already calculated the linear constraints

$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.1})| \leq |occ((v_{start}, v_1), \pi)| - |occ((v_1, v_2), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.1})| \leq |occ((v_3, v_3), \pi)| + |occ((v_3, v_{end}), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.2})| \leq |occ((v_3, v_3), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,1.2})| \leq |occ((v_3, v_3), \pi)| + |occ((v_3, v_{end}), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,2.0})| \leq |occ((v_1, v_2), \pi)|;$$
$$|occ(v_3, \pi, \mathcal{C}_{v_3,2.0})| \leq |occ((v_3, v_3), \pi)| + |occ((v_3, v_{end}), \pi)|.$$

in Example 4.12. These translate to corresponding IPET constraints

$$f_{v_3,1} \leq f_{(v_{start},v_1)} - f_{(v_1,v_2)};$$
$$f_{v_3,1} \leq f_{(v_3,v_3)} + f_{(v_3,v_{end})};$$
$$f_{v_3,2} \leq f_{(v_3,v_3)};$$
$$f_{v_3,2} \leq f_{(v_3,v_3)} + f_{(v_3,v_{end})};$$
$$f_{v_3,3} \leq f_{(v_1,v_2)};$$
$$f_{v_3,3} \leq f_{(v_3,v_3)} + f_{(v_3,v_{end})}.$$

4.10 Chapter Summary

In this chapter, we have developed a method for reducing pessimism in MBTA.

We have presented context-sensitive IPET as a method for reducing pessimism in MBTA. The method is an extension of standard IPET. Unlike standard IPET, context-sensitive IPET does not rely on a single, constant WCET estimate of each program part, but allows for multiple execution scenarios of each program part, with different associated WCET estimates.

Context-sensitive IPET is a generic estimate-calculation method. To be applied, it must be instantiated. We have presented such an instantiation, which relies on the notion of a *context*.

A context of a given node in a CFG is a collection of structurally similar paths in a CFG, with an additional constraint on the number of times that the given node may appear on any path of the context. For a given node, it is possible to choose a set of contexts that forms a classification of the execution times of that node. Such a set of contexts can then be used as execution scenarios in context-sensitive IPET.

We have developed a formal framework for working with contexts. Furthermore, we have presented an algorithm for finding suitable context sets, and demonstrated its correctness.

CHAPTER 5

Reducing Optimism

Notice: The results presented in this chapter on optimism have mostly been obtained through the research effort of Sven Bünte, as part of our research collaboration within the FORTAS *project. The author of this thesis provides this overview for the sake of completeness, but does not claim any scientific results in this chapter as his own.*

In this Chapter, we describe a method for reducing optimism in MBTA.

In Section 5.1, we identify measurements as source of optimism in MBTA. In Section 5.2, we present FROO, an input-data generation technique that seeks to reduce the optimism of MBTA through an increase of the measurement coverage of temporal behavior. In Section 5.3, we present an evaluation for FROO. In Section 5.4, we describe some limitations of FROO.

5.1 Measurements and Optimism

In MBTA, the local WCET estimate for each individual program part is based on measurements. These measurements usually cannot cover all possible initial states for the individual program parts, for two reasons: Firstly, the size of the relevant state space, i.e., the space of possible initial conditions with respect to the timing-relevant computer state (cf. Section 3.3) is usually prohibitively large. Secondly, we want MBTA to be portable, so we cannot allow the method to depend on the specific state space of some particular target hardware.

The simplest possible local WCET estimate for a program part is its maximal observed execution time, i.e., the maximum over all execution times that have been measured for that program part. Such estimates may, however, lead to a global WCET estimate of the complete program that is lower than the program's actual WCET. Figure 5.1 illustrates this by a simple examples.

Optionally, a safety margin can be added to each local maximal observed execution time, or extreme value theory [Col01] can be applied to the collected samples. All such extra effort serves the goal of reducing the amount of underestimation of the local WCET estimates. An alternative approach are techniques that increase the measurement coverage of temporal behavior. In this chapter we present such a technique.

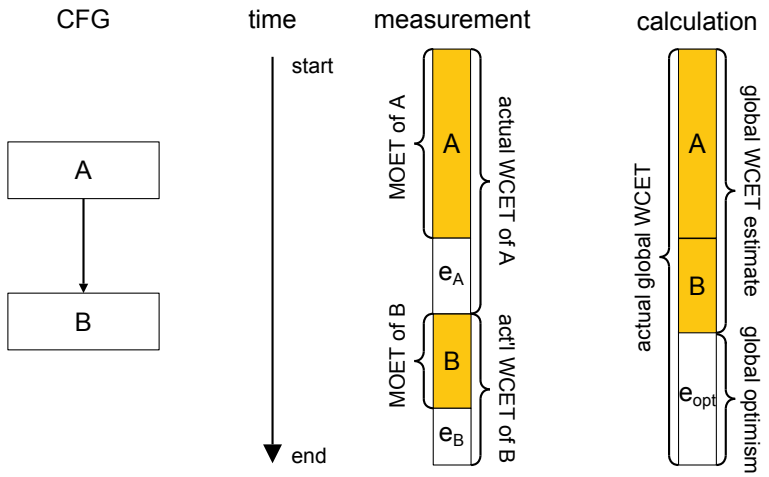

Figure 5.1: Optimism: Using local MOETs instead of the actual WCETs introduce underestimation in the global WCET estimate. Based on the same code example as is Figure 4.1, we assume that the actual WCET of program part A is by an error value e_A higher than the observed MOET. Likewise, the actual WCET of program part B is higher than the observed MOET, by an error value e_B. If the local MOETs are used as local WCET estimates, a simple version of IPET will yield a global WCET estimate that is lower that the actual global WCET, by an error value e_{opt}. In this simple example, we have $e_{opt} = e_A + e_B$.

Recall that the goal of performing execution-time measurement in MBTA is to infer a worst-case execution-time estimate for each program part (cf. Section 3.6). Furthermore, we have assumed that each program part corresponds to a particular node $v \in V$ in the program's CFG $\mathcal{G} = (V, E, v_{start}, v_{end})$ (cf. Section 3.8).

Let $\mathcal{D}$ be the domain of all possible input vectors that can be fed to the given program. We assume the availability of a measurement method (cf. Section 3.15) that yields a timed trace $\pi_d = (v_1, t_1) \dots (v_n, t_n)$ for each run of the program, under input vector d, and on a particular execution platform of interest[1].

We first define what we mean by the term *test suite*:

Definition 5.1. *A* test suite Γ *is a set of input vectors, i.e.,* $\Gamma \subseteq \mathcal{D}$.

The maximal MOET of a node over a test suite is given by the following definition:

Definition 5.2. *The* maximal MOET $moet_{v,\Gamma}$ of some node v over a test suite Γ *is the maximal MOET of* v *in any timed trace* pi_d*, for* $d \in \Gamma$*, i.e.,*

$$moet_{v,\Gamma} = \max\{moet_{v,\pi_d} \mid d \in \Gamma\}.$$

[1]Note that the obtained timed trace π_d might differ between multiple measurement runs with the same input vector, since the input vector only partially specifies the initial state of the computer (cf. Section 3.3).

5.2 FROO

FORTAS Reduction of Optimism (FROO) is an input-data generation technique that seeks to reduce the optimism of MBTA through an increase of the measurement coverage of temporal behavior. The technique is based on the principles of genetic algorithms and model checking [CGP00], and tries to achieve its goal through a joint increase of the MOET over all nodes.

The motivation for using a genetic algorithm stems from our previous work [BZTK11], where we demonstrated that random input-data generation is very fast, but unable to cover all parts of a program. Model checking, on the other hand, can be used to generate test suites that satisfy sophisticated structural coverage criteria, but requires considerable computational resources.

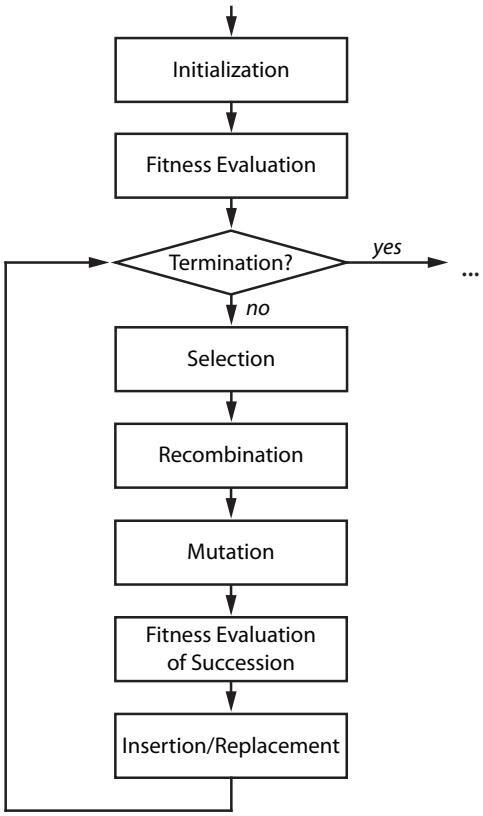

Figure 5.2: The FROO workflow.

FROO proceeds according to the standard phases of a genetic algorithm (cf. Figure 5.2):

FROO starts with an initial test suite that covers all reachable nodes, thereby guaranteeing that each reachable node is exercised at least once. Our implementation uses FShell [HSTV08, HTVS10, HSTV11] to generate appropriate input vectors for this initial test suite. The initial test suite forms the seed population of a genetic algorithm that tries to further improve the quality of the test suite by heuristic optimization. In the following we describe the individual steps of FROO:

Initialization: The individuals of the genetic algorithm are input vectors, and the complete population forms a test suite. We write Γ_i to denote the population; Γ_0 is the seed population, which we generate in the following way: First, we generate a test suite Γ_B that satisfies node coverage using FShell. The input to FShell is a C program and a test suite specification, expressed in the FQL [HSTV11]. For example, if our nodes are basic blocks, we use an FQL queries of the form

```
IN @FUNC(foo) cover @BASICBLOCKENTRY.
```

The seed population must include test suite Γ_B. However, if Γ_B contains only a few input vectors, we add a supplement to random input vectors in the seed population.

Using a seed population that satisfies node coverage is superior to using an initial population of random input vectors, because it incorporates knowledge on how to reach each of the nodes. On the other hand, random test suites lack this knowledge (cf. Figure 3.8). Moreover, using a seed population that satisfies node coverage guarantees at least one measurement of each reachable node.

Fitness Evaluation: We use a fitness function that considers the execution time distance to the maximal execution time, observed with respect to the current population. In order to account for jitter, and to even further promote distances that are close to the MOET, we square the distance:

$$F_{\Gamma_i}(d) = \sum_{v \in V} \frac{1}{1 + (moet_{v,\Gamma_i} - moet_{v,\pi_d})^2}, \text{ for } d \in \mathcal{D}.$$

Selection: Inspired by the work of Khan and Bate [KB09], we use roulette-wheel selection and pick 50% of the individuals in the current population for recombination. We determined the particular ratio through experimentation.

Recombination: As suggested in previous work on input data generation via genetic algorithms [Ata03], we use single-point crossover.

Mutation: We set the mutation probability to 2%, i.e., each individual in the offspring has a chance of 0.02 to be mutated at one randomly chosen variable. For a variable that is chosen to be mutated, a new uniformly distributed value in the variable's domain is generated. We determined the particular mutation probability through experimentation.

Insertion/Replacement: We use elitist reinsertion: the whole offspring is inserted into the current population; those individuals showing the lowest fitness are replaced.

Termination: Our genetic algorithm terminates after a fixed number of iterations.

5.3 Evaluation

In this section we evaluate FROO. We propose the following metric for quantifying the optimism of a given test suite, where higher values indicate lower pessimism:

Definition 5.3. *The* average MOET $moet_\Gamma$ of a test suite Γ *is the arithmetic mean over the MOETs of all nodes* $v \in V$ *over* Γ*, i.e.,*

$$moet_\Gamma = \frac{1}{|V|} \cdot \sum_{v \in V} moet_{v,\Gamma}.$$

This metric is based on the observation that a high MOET of an individual node is less optimistic than a low MOET for the same node. However, to be able to compare the optimism of different test suites for the same given program, we would like to have a measure that summarizes the optimism of all nodes. By using the arithmetic mean over all nodes, we make no assumption about how the global WCET estimate is derived during estimate calculation. However, it would be possible to used other metrics that are tailored to a particular estimate calculation methods. If IPET is used, for example, it might be useful to consider a metric that favors nodes that occur frequently on the WCET path.

We note that the given metric is only useful for comparing the optimism of test suites for the same given program, but not so for comparing the optimism of test suites of different programs.

Table 5.1: Average MOET $moet_\Gamma$ for a test suite of random input vectors (RDG), a test suite satisfying node coverage (NC), a test suite obtained with FROO using a seed population including individuals obtained by model checking (FROO), and a test suite obtained with FROO using a seed population containing only random input vectors (FROO*). MAX denotes maximum over all average MOETs obtained in any experiment.

Benchmark	RDG	NC	FROO	FROO*	MAX
md-binary_search-binary_search	55.69	49.64	65.93	56.36	66.93
md-bsort10-BubbleSort	84.44	69.17	85.47	85.47	86.80
is-engine_control_cs1-AktuatorMotorregler	44.66	46.06	61.19	56.33	63.67
jop-lift_control-ctrl_loop	23.80	51.35	61.31	24.84	66.82
pb-a1-course_run	53.86	48.15	60.20	42.20	60.20
pb-a1-course_pid_run	61.30	54.41	77.76	76.35	82.06
pb-a2-atan2	84.48	87.33	105.33	104.93	111.07
pb-a2-sin	52.93	50.21	73.52	52.89	74.48
pb-a2-navigation_update	44.96	40.00	54.80	36.00	54.80
pb-a2-nav_home	74.80	77.31	96.90	93.00	98.81
pb-a2-compute_dist2_to_home	69.58	64.83	89.50	44.12	89.50
pb-f1-check_mega128_values_task	50.19	54.91	69.09	27.27	71.00
pb-f1-servo_set	0	79.70	87.44	0	91.35
pb-f2-vector_10	88.54	85.61	99.00	76.06	101.56

Table 5.1 gives an overview of the results of our evaluation. Details on each of the used benchmarks, as well as on the experimental setup can be found in Chapter 7, where we present the evaluation of the complete the FORTAS tool.

TODO: rewrite this

Column one (Benchmark) indicates the name of each benchmark.

Column two (RDG) shows $moet_\Gamma$ for 1000 randomly generated input vectors. An extreme case that we observed is `pb-f1-servo_set`, where the whole function is never reached by the input data (it is conditionally called by `pb-f1-check_mega128_values_task`).

Column three (NC) shows $moet_\Gamma$ for a test suite that satisfies node coverage. It is not possible to favor any of the two methods in terms of $moet_\Gamma$.

Column four (FROO) shows $moet_\Gamma$ for FROO after 25 iterations.

Column six (MAX) shows $moet_\Gamma$, with Γ including all input vectors we generated during all experiments for the respective benchmark.

We see that $moet_\Gamma$ is consistently higher for FROO than for both RDG and NC. Also, its values are close to $moet_\Gamma$ for MAX. The results also show that FROO benefits from the initial test suite satisfying node coverage. FROO* (results shown in column five) uses the same configuration as FROO, but the former uses a seed of 200 randomly generated input vectors, i.e., it does not utilize model checking. The results in Table 5.1 show that using model checking in generating the seed population pays off: FROO's average WCET estimate is greater or equal than FROO*'s for all benchmarks. A detailed evaluation of FROO is provided in [BZK11].

5.4 Limitations

FROO requires an initial test suite that satisfies node coverage, such that at least one execution-time measurement, as well as that corresponding input vector are available for every node. In our implementation, we use FShell to generate an appropriate test suite—this works fine for most of the benchmarks that we analyzed during our evaluation. We could generate appropriate test suites for most of the benchmarks within a few seconds. However, two of the benchmarks posed a challenge:

For benchmark `md-binary_search-binary_search`, as well as for Problem A3 from the PapaBench benchmark suite, FShell failed to generate a seed population within the given time frame, so we had to prematurely stop the respective experiments. In the former case, the problem was due to general scalability issues of model checking. As a workaround, we reduced the size of the input vector from 100 to 25 elements. In the latter case, the problem had been caused by the used floating point operations. In this case, the underlying SMT solver of the CBMC model checker failed to find a suitable solution within the given time frame.

An inherent limitation of MBTA is that the initial state of the target system before measurement cannot be controlled completely, as the hardware is not known in full detail. This is another source for optimism. We did not address this problem specifically, but there exists work on how WCET analysis can be complemented with mechanisms that enforce a defined hardware state. For example, some architectures support cache invalidation. As another example, the pipeline can be filled with NOP instructions to bring it (including its branch predictor) into a defined state [KKP10]. However, those techniques require some knowledge about the target hardware and are not subject to our current research.

5.5 Chapter Summary

In this Chapter, we have considered a method for reducing optimism in MBTA. We have presented FROO, an input-data generation technique that seeks to reduce the optimism of MBTA through an increase of the measurement coverage of temporal behavior. We

have described how FROO works, and we have provided an evaluation of the technique.

CHAPTER 6

FORTAS: High-Precision MBTA

In this chapter, we present the FORTAS approach for high-precision MBTA.

In Section 6.1, we present requirements for an industrial-strength WCET analysis tool. These requirements—which had been collected in a previous study—formed the initial motivation for our research on high-precision MBTA. We also present additional requirements that we identified as essential for an MBTA-based industry-strength WCET analysis tool.

In Section 6.2, we review the features that distinguish the FORTAS approach from standard MBTA methods. One particularly innovative feature is iterative refinement, which we review in Section 6.3. In Section 6.4, we review the modular software architecture of the FORTAS tool.

6.1 Requirements

The FORTAS tool is a prototypical implementation of the FORTAS approach, intended to integrate smoothly into the engineering process of embedded software. Previous work on MBTA [KPW04] has established the following four requirements for an industrial-strength WCET analysis tool:

Requirement IR-1: The tool must work with little user interaction. In particular, it must not be expected that users of the tool provide manual code annotations about possible and impossible execution paths of the code. For example, when model-based development tools like Matlab/Simulink are used to design the system, the WCET analysis tool must be able to extract such information by analyzing the code generated by the code generator of Matlab/Simulink.

Requirement IR-2: The method must integrate into the development tool-chain of customers without requiring a modification of the tool-chain (for example, modifications of Matlab/Simulink, the code generator, or the C compiler). However, it is permissible to restrict the use of the tool-chain to enable WCET analysis. For example, the available application development features of model-based development tools like Matlab/Simulink may be restricted, or certain compiler optimizations may be deactivated.

Requirement IR-3: The method must be adaptable to changes in the tool-chain without expensive modification.

Requirement IR-4: The WCET analysis method must be easily retargetable to different hardware settings, i.e., the implementation or configuration effort must be economically feasible.

The FORTAS tool is a prototypical implementation of the FORTAS approach that has been designed to fulfill all of these industrial requirements. However, since the FORTAS tool is a research prototype without commercial-scale development resources, we had to restrict the extent of our engineering effort. The current fulfillment status of the industry requirements is as follows:

Fulfillment status of IR-1: Currently, the FORTAS tool requires the user to manually specify loop iteration constraints. There are existing third-party tools that are able to infer such constraints automatically in many common cases (cf. Section 3.8). Integrating such components into the FORTAS tool is a matter of engineering.

A general and fully automatic method of inferring loop iterations is currently unknown, so the need for user annotations cannot be fully eliminated. State-of the art tools, however, can keep the annotation effort low. In particular, a fully automatic analysis of source code that is automatically generated by code generators like Matlab/Simulink may be feasible [KLFP02].

Fulfillment status of IR-2: For the FORTAS tool, we have chosen the GNU-based TriCore C/C++ tool-chain from HighTec EDV-Systeme GmbH [Hig11] as our target tool-chain. In principle, however the FORTAS tool could be used with any other compiler that can produce code for the Infineon TriCore TC1796. The FORTAS tool requires a tight correspondence between the source code of the software under analysis and its corresponding executable machine code. To achieve such a correspondence, most compiler optimizations must be deactivated for WCET analysis (cf. Section 3.6).

Fulfillment status of IR-3: The FORTAS tool meets this requirement, as it allows, in principle, for the use of an arbitrary compiler tool-chain, as discussed before. The possibility to easily adapt the tool suite to a new release of a certain tool-chain can be seen as special case of adapting the FORTAS tool to an entirely new tool-chain.

Fulfillment status of IR-4: The FORTAS tool includes a measurement driver for the Infineon TriCore TC1796 processor. More details on the current setup can be found in Sections 7.1 through 7.3. Due to the general analysis method of the FORTAS tool, and also due to its modular software architecture, an adaption to other target platforms is expected to be relatively straightforward, through the implementation of a target-specific measurement-driver (cf. Section 3.15).

A crucial issue in MBTA is the closeness of the obtained WCET estimates (cf. Section 1.6). However, closer WCET estimates may come at the cost of a higher resource

burden at analysis time, including a longer analysis time. The optimal balance between closeness of the obtained WCET estimates and analysis effort may vary from use case to use case. For example, a quick response from the analysis might be preferable at an early development stage, whereas closer estimates might be required later on, as the design matures. We therefore would like the user to be able to take control over this balance, be way of the following requirement:

Requirement FO-1: There must be a way for the user to adjust the balance between the achievable closeness in the WCET estimate and the required analysis effort.

Besides adjusting the balance between estimate closeness and analysis effort, the user is also likely to be interested in adjusting the balance between pessimism and optimism. In accordance, we have imposed the following requirement:

Requirement FO-2: There must be a way for the user to adjust the balance between the pessimism and optimism of the analysis results.

Figure 6.1 illustrates how the balance between closeness and computational effort on the one hand, and the balance between optimism and pessimism on the other hand form the basis of a two-dimensional space of analysis configurations.

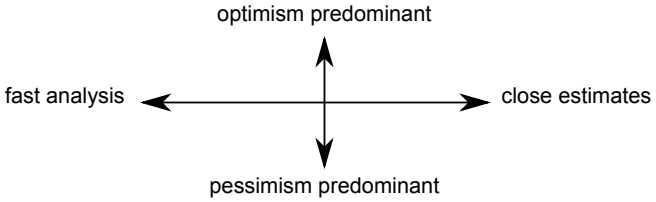

Figure 6.1: Refinement control.

The FORTAS tool was designed to support agility under changing system specifications and requirements. This is particularly important for application in design space exploration (cf. Section 1.5). We have imposed the following requirement:

Requirement FO-3: If the software or target platform under investigation is changed, results from previous analysis runs should be reused, as far as possible.

The FORTAS tool was designed for extensibility and scalability. The corresponding requirements are:

Requirement FO-4: Analysis services can be used concurrently by multiple users in different locations.

Requirement FO-5: The analysis must scale with the available resources. Additional resources (computing power, data storage, additional target hardware, etc.) can be added in order to boost the analysis.

We have tried to fulfill these requirement to the extent possible in a research project. The following list explains the fulfillment status of each requirement:

Fulfillment status of FO-1: The FORTAS approach introduces the novel feature of adaptive input-data generation, in order to improve the precision of MBTA. Adaptive input-data generation is implemented as an iterative refinement approach (cf. Section 6.2). Through the addition of this new feature, it becomes possible to increase the closeness of the obtained WCET estimate, at the cost of a higher resource consumption during analysis.

Fulfillment status of FO-2: The FORTAS approach integrates orthogonal methods to reduce and pessimism (cf. Chapter 4) and optimism (cf. Chapter 5), which work together to yield close WCET estimates. It is possible to adjust the balance between optimism and pessimism by balancing these methods through parametrization.

Fulfillment status of FO-3: The FORTAS tool provides a data repository, where all relevant information from previous analysis runs, like input data, timed traces, etc., is persistently stored. This information can be reused later. Even if the software under analysis or the target hardware are changed, it is possible to reuse some available information: For example, if the target processor is replaced, a large amount of input vectors can usually be reused. Moreover, test cases from functional testing can be imported or exported with little effort.

Fulfillment status of FO-4: The FORTAS tool provides distributed services and can handle concurrent requests from a single as well as from multiple users (cf. Section 6.4). The requests are implemented as remote procedure calls.

Fulfillment status of FO-5: The FORTAS tool is based on a modular software architecture (cf. Section 6.4), where each module is responsible for providing a certain set of services. Service requests to different modules can be parallelized. Moreover, multiple instances of each module can be created to process concurrent requests in parallel, provided that enough resources are available. The communication between modules is implemented by remote procedure calls, which enables the distribution of module instances over multiple host computers. Shared data is managed centrally by a special data-repository components.

6.2 Distinctive Features of the FORTAS Approach

The FORTAS approach extends standard MBTA (cf. Section 3.13) by introducing multiple new operations into the data-flow, as illustrated in Figure 6.2:

Tracing measurement: Unlike traditional MBTA, which can operate on unspecific (i.e.: context-free) execution times of individual program parts, the FORTAS approach makes use of the contextual information contained in timed traces. In particular, the FORTAS integrates methods for reducing pessimism that rely on the control information contained in the timed traces. To make clear that timed traces are required, rather than merely separate measurements of individual program parts, we call the corresponding measurement operation *tracing measurement*.

Context-sensitive estimate calculation: This operation uses the timed traces that are generated during tracing measurement and generates a global WCET estimate. The important difference to estimate calculation in standard MBTA is that context-sensitive estimate calculation applies methods that make use of the contextual information contained in the timed traces, in order to produce less pessimistic WCET estimates. The FORTAS tool uses context-sensitive IPET (cf. Section 4.4).

Adaptive input-data generation: This operation generates test suites to be used during tracing measurement. The important difference to input-data generation in traditional MBTA is that adaptive input-data generation is intended for iterative input-data generation, based on feedback from previous measurement results and from context-sensitive estimate calculation. The FORTAS tool integrates FROO (cf. Section 5.2), in order to produce less optimistic WCET estimates. Moreover, it also uses information about the execution scenarios within the context-sensitive estimate calculation. This allows adaptive input-data generation to generate test suites that are tailored for individual execution scenarios. To produce additional test suites for specific execution scenarios is important, because the implicit splitting of the test suite along execution scenarios that takes place in context-sensitive estimate calculation might leave certain execution scenarios poorly covered by test vectors. This is compensated by generating test suites for specific, poorly covered, execution scenarios.

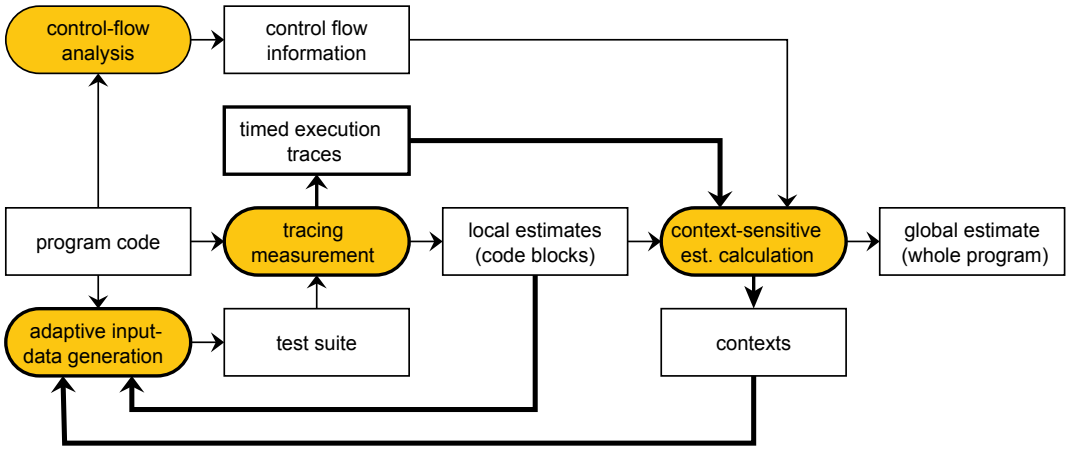

Figure 6.2: The extended workflow of the FORTAS approach.

6.3 Iterative Refinement

Standard MBTA features a linear workflow: A program of interest is provided as input to the analysis tool, the timing analysis is performed, and an estimate of the WCET is produced after a finite amount of time.

The workflow adopted by the FORTAS approach is very different from this: Once a program of interest has been submitted to the analysis process, an iterative refinement loop is started that generates an ongoing stream of test suites that are used for new measurements. The resulting stream of new timed traces is buffered, and, periodically, a new estimate calculation is performed. Adaptive input-data generation makes use of feedback data from measurement and estimate calculation to guide the generation of new test suites. Over time, this ongoing process yields more and closer intermediate WCET estimates.

At any point in time, the user is able to inspect the current WCET estimate. However, there is no clearly defined end of the analysis process. The user may let the process continue until he feels comfortable with the achieved precision, or he might specify an exit condition for the process, like some upper bound on resource usage, or an upper bound on the number of refinement steps to be performed.

6.4 The Software Architecture

The FORTAS tool embraces an extensible, modular design philosophy that simplifies maintainability, scalability, and portability, through two major strategies:

- easy replacement of individual modules, for example, to provide support for new target platforms;
- easy parallel and distributed processing, for example, through the distribution of input-data generation, measurement, and estimate calculation with multiple workers for different code fragments.

MBTA requires the cooperation of different tasks with different resource requirements:

Control-flow analysis: The resource requirements of control-flow analysis depend on the complexity of the analyses that are performed. Whereas a CFG and simple loop bounds can usually be obtained at relatively small costs in computation time and memory, more sophisticated analyses of the control flow might be more demanding. However, these methods are less demanding than the brute force methods used in input-data generation. Sharing the computational infrastructure with global estimate calculation might be considered.

Input-data generation: Input-data generation can require large amounts of processing power, in particular if model checking is applied. Model checking can, moreover, demand a lot of system memory. Even in a small setup, a powerful machine with a large amount of memory is required for this task. For larger setups, a small server farm should be considered, where model checking can be performed in parallel, on different subspaces of the entire state space.

Measurement: Unless the measurements are performed on a cycle-accurate simulator, access to the target platform is required. Depending on the setup, performing the measurements sequentially can impose a performance bottleneck. Luckily, this task can easily be parallelized by splitting up the test suite. Using multiple instances

of the target platform in parallel can thus speed up the measurement process linearly. However, certain measurement setups can be quite costly and might not be available on all sites, so remote sharing of the measurement setup might be necessary, especially if multiple users need to access them just on an occasional basis. A pool of different target platforms might be considered, if a comparative analysis is required. Moreover, a suitable infrastructure for preprocessing, relaying, storing, and retrieving large amounts of tracing results must be provided.

Estimate calculation: Like in control-flow analysis, the resources requirements of estimate calculation depend on the actual method used. In the case of IPET, an integer linear programming problem must be solved. Whereas ILP problems are known to be NP-hard in theory, we never ran into performance issues with IPET. Sharing the computational infrastructure with control-flow analysis should be considered.

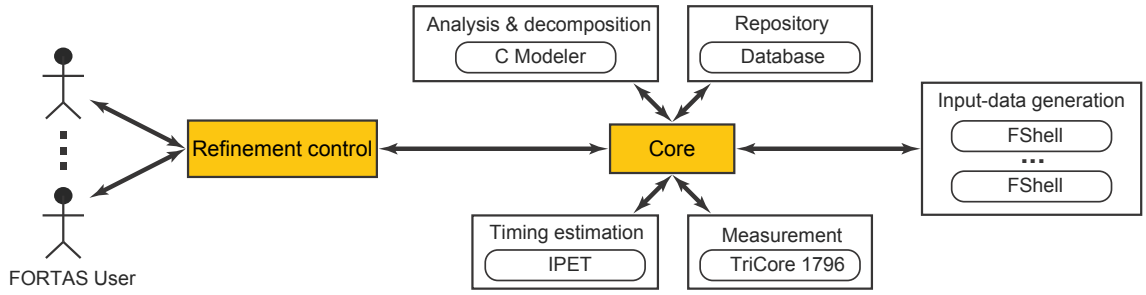

Figure 6.3: The software architecture of the FORTAS tool.

As can be seen in Figure 6.3, the high level software architecture of the FORTAS tool is based on a collection of replaceable modules that are centered around a core service that is responsible for coordination, communication, and job distribution. The following is a description of the individual components of the software architecture:

Analysis and decomposition: provides the CFG of the program under scrutiny.This module can run program analyses, for example, to derive loop iteration bounds. Such analysis are not currently implemented, though. Loop iteration constraints must currently be annotated manually.

Input data generation: provides services for generating test data, given either a specification of a test suite or a search heuristic. The FORTAS tool uses *FShell* [HSTV08, HTVS10, HSTV11]. Multiple FShell workers run in parallel to generate input data at a high rate.

Measurement: provides services for executing the program under analysis on the target hardware and yields timed traces. If necessary, the module also performs compilation and linking for the target platform. We currently support tracing on the TriCore 1796 microprocessor via a Lauterbach LA-7690 PowerTrace device.

Repository: provides persistent storage and retrieval services for source code, binaries, timing models, CFGs, test suites, timed traces, and WCET estimates. The FORTAS tool is based on the concepts presented in [BT08] as a PostgreSQL database.

Estimate calculation: provides services for calculating a global WCET estimate from the local WCET estimates of individual code blocks. The FORTAS tool provides standard and context-sensitive IPET as calculation methods.

The Core: takes care of both, intercommunication between modules and parallel processing.

Refinement control: provides the interface for the user.

Modules are independent of each other in the sense that *jobs* assigned to some module A can be processed in parallel with the jobs assigned to another module B, since different modules do not share critical resources.

An example of a job is the generation of input data according to a given test suite specification. Jobs are defined in a way that only a single module is needed for processing them.

An instance of a module is called a *worker*. Some representative examples of workers are given in the oval boxes of Figure 6.3).

Multiple workers can be spawned to increase the job throughput of a given module. The *repository* module, which maintains a central storage and retrieval service for analysis information, is an exception, however. Only one *database* worker is currently used, to make keeping the database easy. In the FORTAS tool, this centralized solution proved sufficient from a performance point of view. For high-performance setups, a distributed database may be more suitable. We have not further investigated such a distributed solution.

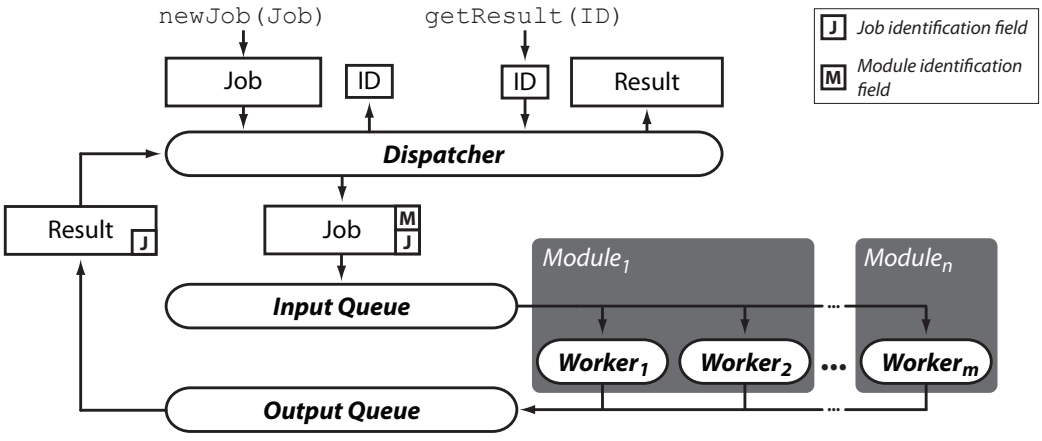

Figure 6.4: Asynchronous job processing.

The FORTAS tool supports concurrent job processing, as depicted in Figure 6.4. As a first processing step, the *Dispatcher* assigns a unique job identifier and a unique module domain to each incoming job. The ID is immediately returned to the callee. The subsequent call `getResult`, however, blocks until all processing stages of a job are completed and a result for the provided job identifier is available. After dispatching, a job gets enqueued for a worker that fits the job's target module. The dispatcher and workers operate asynchronously; each worker W_i performs the following:

1. Check input queue for a job that fits to W_i;
2. Move that job from the input queue to W_i;
3. Process job;
4. Put result and job identifier into output queue;
5. Continue with (1).

Once a result is in the output queue, the dispatcher forwards it to the correct callee.

6.5 Chapter Summary

In this chapter, we have presented the FORTAS approach for high-precision MBTA.

Both, the FORTAS approach and the FORTAS tool, have been designed according to industrial requirements. These requirements—which had been collected in a previous study—emphasize a need for little user intervention, easy integration into existing development tool-chains, and portability. A comparison of static WCET analysis and the FORTAS approach reveals the latter as more suitable basis for fulfilling these requirements. Further requirements that we have identified refer to user control over the balance between pessimism, optimism, and resource usage, reuse of previous analysis results, concurrent use by multiple users, and scalability.

The FORTAS approach introduces several innovative features, which include tracing measurement, context-sensitive estimate calculation, and adaptive input-data generation. Moreover, the FORTAS tool embraces an extensible, modular design philosophy that simplifies maintainability, scalability, and portability.

CHAPTER 7

Experiments

In this chapter, we present an experimental evaluation of the FORTAS approach.

We first describe the target platform that we used in our evaluation. In Section 7.1, we present the TriCore TC1796 microprocessor. In Section 7.2, we present the OCDS debugging interface, which provides the technical prerequisites for collecting timed execution traces without exerting a probe effect on the system. In Section 7.3, we present the TriBoard TC179X Evaluation Board that we used as a platform for the TC1796. In Section 7.4, we present the benchmarks that we used. In Section 7.5, we explain the design of our experiments. In Section 7.6, we present the results of our evaluation.

7.1 The TriCore TC1796

We have chosen the Infineon TriCore TC1796 as target processor for the prototype implementation of the FORTAS approach.

The TC1796 is a fairly complex 32-bit microprocessor—targeted at the automotive market—that provides numerous on-chip peripherals that are useful for embedded applications. These peripherals include a *Micro Second Bus*, a fast analog-to-digital converter unit, and a *Micro Link Interface*. The processor is specifically targeted at the automotive industry, and features simple versions of many of the performance-enhancing features found in modern desktop and server processors, like caching, pipelining, and branch prediction. Contrary to popular belief, the TriCore TC1796 has only a single processing core. It, however, features three parallel instruction pipelines that allow parallel processing of different types of instructions, as well as a separate floating point unit.

Even though the FORTAS approach does not rely on knowledge about the internal operation of the target microprocessor, we believe that such an understanding may be useful to you, the reader of this thesis, for the interpretation of our experimental results. We therefore provide an overview of three features that we consider particularly relevant for WCET analysis: the memory subsystem, the instruction pipeline, and branch prediction. More details about the TriCore architecture and the TC1796 microprocessor can be found in the corresponding technical manuals [Inf06, Inf07].

Figure 7.1 provides a high-level view on the structure of the bus systems of the TC1796 processor: The basic design is based on a Harvard architecture, with sepa-

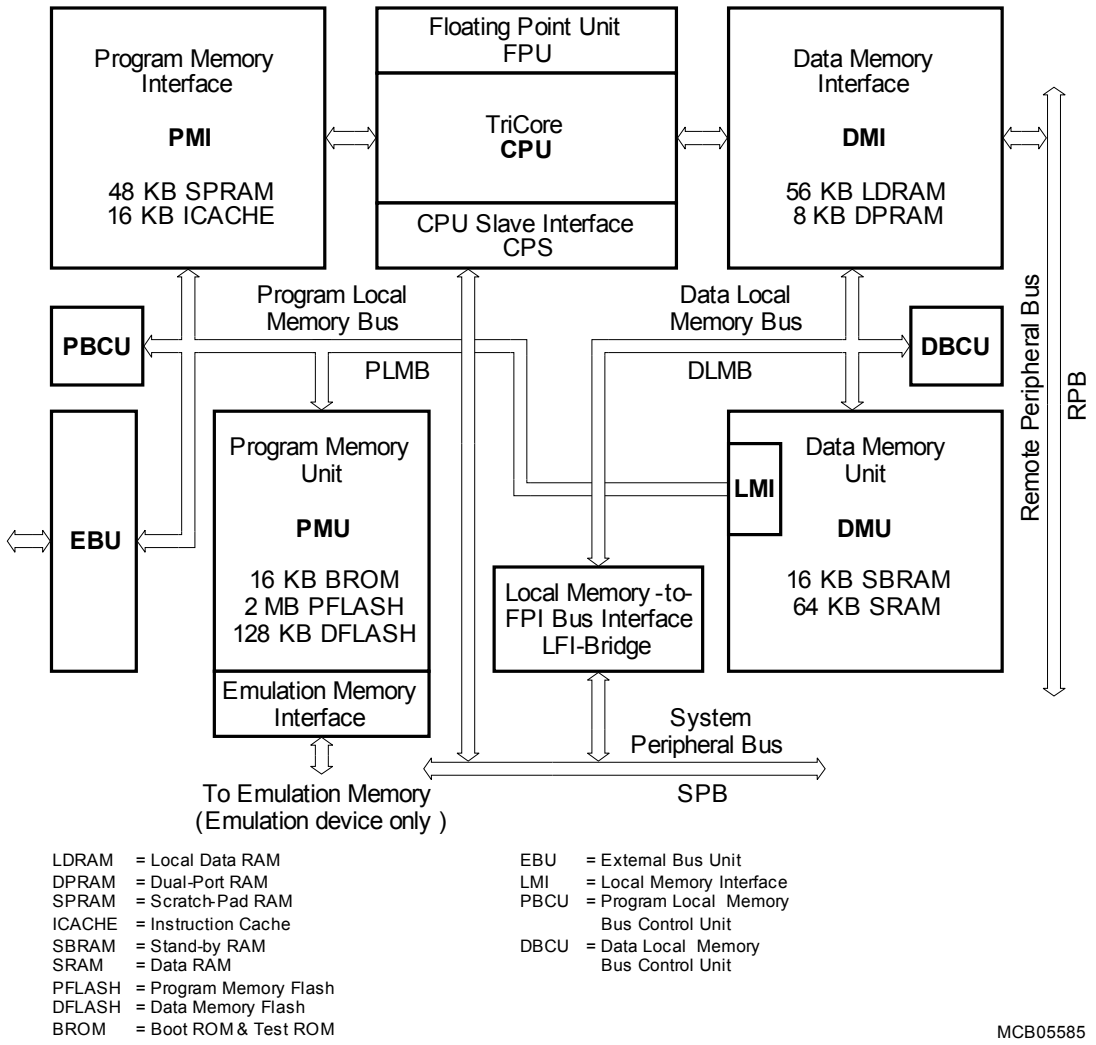

Figure 7.1: Block diagram of the bus systems of the TC1796. Illustration taken from [Inf07].

rate interfaces for program and data memory (PMI, DMI). On the whole, the memory subsystem is fairly complex and allows for an abundance of different memory configurations that can be chosen by the system designer. The system consists of the following components:

Program memory interface (PMI): The PMI (cf. Figure 7.2) is directly connected to the CPU and is responsible for all accesses to program memory. It is equipped with 64KiB of RAM, of which 16KiB can be used as instruction cache (ICACHE) and of which 48KiB can be used as scratchpad memory (SPRAM). The ICACHE is a two-way set-associative LRU cache with a line size of 256 bits, and a validity granularity of 4 double-words (64 bit). The ICACHE can be globally invalidated to provide support for cache coherency. The ICACHE can be bypassed, to provide direct access to the program local memory-bus (PLMB). The CPU interface supports unaligned, i.e., 16-bit aligned, accesses with a penalty of one cycle for

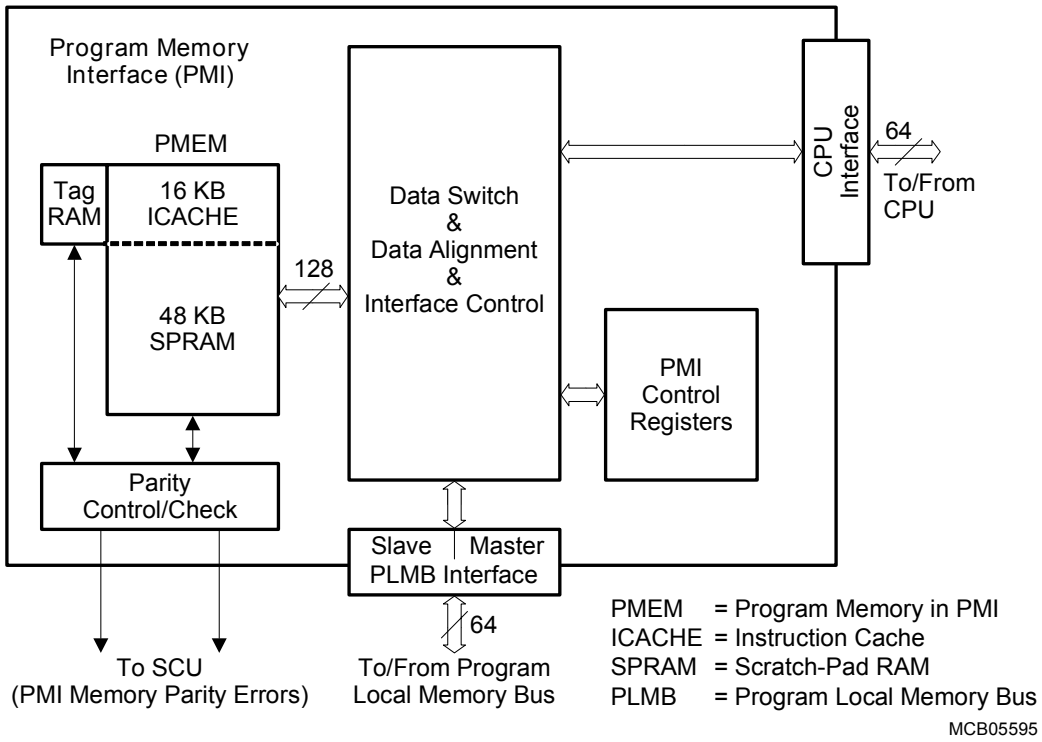

Figure 7.2: Block diagram of the TC1796 program memory interface (PMI). Illustration taken from [Inf07].

unaligned accesses that cross cache lines.

Data memory interface (DMI): The DMI (cf. Figure 7.3) is directly connected to the CPU and is responsible for all accesses to data memory. It is equipped with 64KiB of RAM, 8KiB of which is dual-port RAM (DPRAM) that is accessible from the CPU and from the remote peripheral bus (RPB), and of which 56 KiB is local data memory (LDRAM). The CPU interface supports unaligned, i.e., 16-bit aligned, accesses with a minimum penalty of one cycle for unaligned accesses that cross cache lines. There is a directly accessible interface to the data local memory-bus (DLMB) that provides access to the rest of the system.

Program local memory-bus (PLMB): The DLMB is a synchronous, pipelined bus that connects the DMI to the rest of the data-memory system. The bus protocol supports single transfers of 8, 16, 32, and 64 bits (cf. Figure 7.4), as well as block transfers of 64 bits (cf. Figure 7.5). The PLMB is managed by the program local memory-bus control unit (PBCU), which handles requests from PLMB master devices, which are the PMI and the program memory unit (PMU). Access arbitration takes place in each cycle that precedes a possible address cycle, and is based on the priority of the requesting master device. The PMI has priority over the PMU. Busy slave devices can delay the start of a PLMB transaction.

Data local memory-bus (DLMB): The DLMB is a synchronous, pipelined bus that connects the DMI to the rest of the data-memory system. The bus protocol sup-

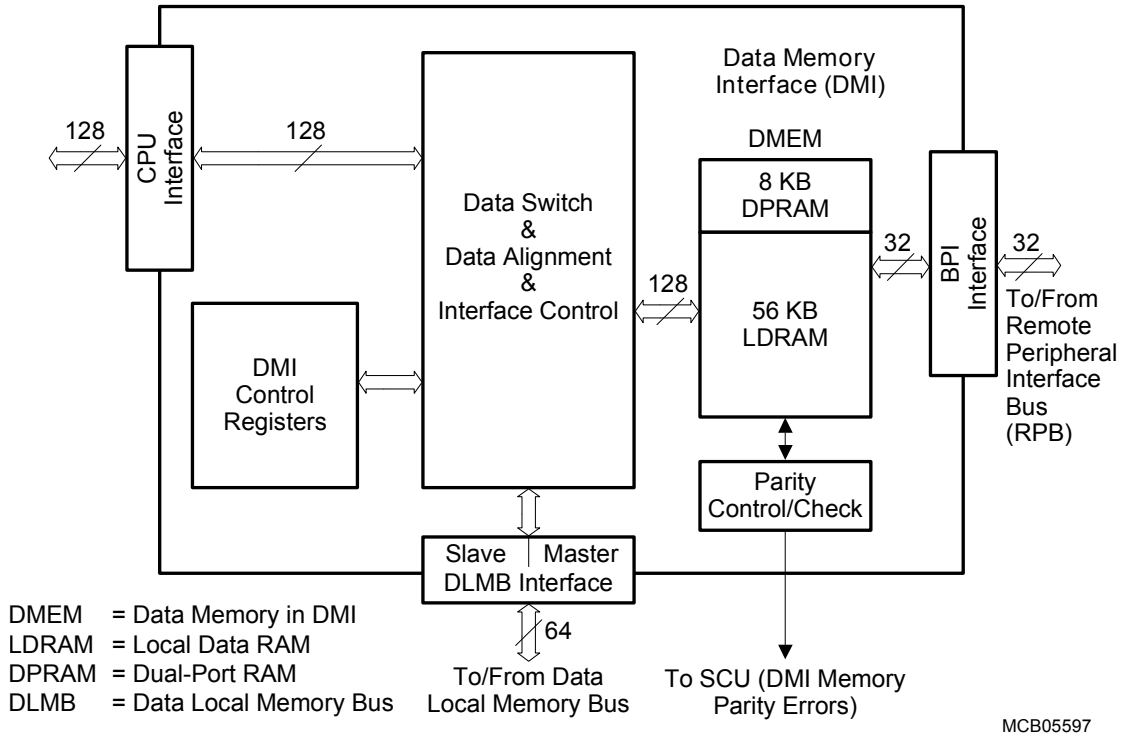

Figure 7.3: Block diagram of the TC1796 data-memory interface (DMI). Illustration taken from [Inf07].

ports single transfers of 8, 16, 32, and 64 bits (cf. Figure 7.4), as well as block transfers of 64 bits (cf. Figure 7.5). The DLMB is managed by the data local memory-bus control unit (DBCU), which handles requests from PLMB master devices, which are the DMI and the data-memory unit (DMU). Access arbitration takes place in each cycle that precedes a possible address cycle, and is based on the priority of the requesting master device. The DMI has priority over the DMU. Busy slave devices can delay the start of a DLMB transaction.

Program local memory-bus control unit (PBCU): The PBCU is responsible for managing data transfers on the PLMB.

Data local memory-bus control unit (DBCU): The PBCU is responsible for managing data transfers on the DLMB.

Program memory unit (PMU): The PMU (cf. Figure 7.6) is connected to the PLMB. It is equipped with 2MiB of program flash memory (PFLASH), 128KiB of data flash memory (DFLASH), and 16KiB of boot ROM (BROM).

Data memory unit (DMU): The DMU (cf. Figure 7.7) is connected to the DLMB. It is equipped with 64KiB of SRAM and 16KiB of standby memory (SBRAM).

Local memory interface (LMI): The LMI is a part of the DMU. It allows the DMI and the DMU to access the PLMB, thereby enabling data transfers to and from other PLMB devices, like the EBU.

External bus unit (EBU): The EBU (cf. Figure 7.8) is connected to the PLMB and serves as an interface to external memory or peripheral units. It supports asynchronous or burst-mode external accesses. The external bus may be shared with other bus masters. Arbitration can be performed either by the EBU, or by an external bus master.

Local Memory to FPI bridge (LFI bridge): The LFI forms a bi-directional bridge between the DLMB and the peripheral FPI bus.

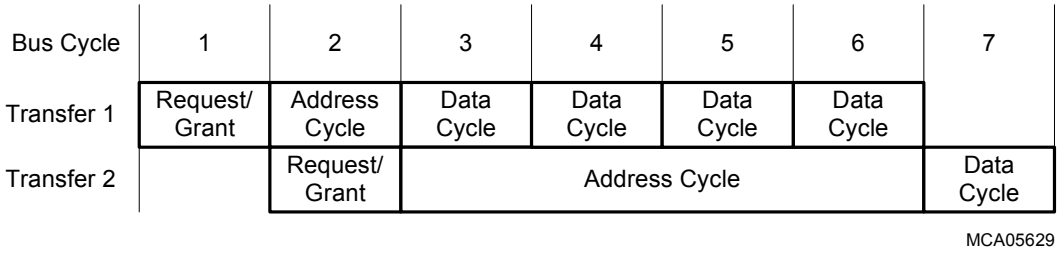

Figure 7.4: Timing of a LMB basic transaction. Illustration taken from [Inf07].

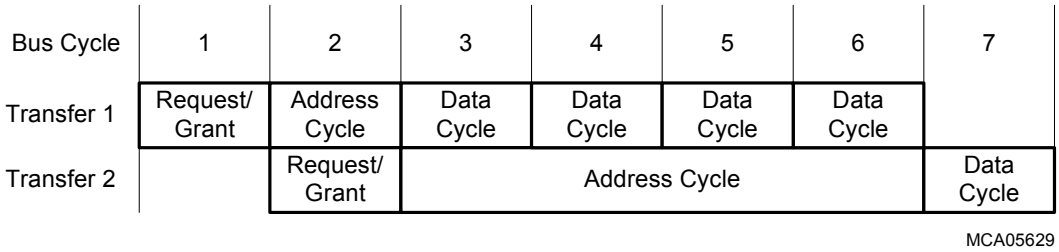

Figure 7.5: Timing of a LMB block transaction. Illustration taken from [Inf07].

Figure 7.9 provides a high-level view of the structure of the TC1796 CPU, which consists of the following components:

Instruction fetch unit: The instruction-fetch unit pre-fetches and aligns incoming instructions from the PMI and issues them to the appropriate instruction pipeline.

Execution unit: The execution unit consists of three parallel pipelines, each of which can process a different type of instructions. The integer pipeline and the load-/store pipeline each consist of the following four stages: fetch, decode, execute, and write-back. The loop pipeline consists of the two stages: decode and write-back. The integer pipeline handles data arithmetic instructions, including data conditional jumps. The load/store pipeline handles load/store memory accesses, address arithmetic, unconditional jumps, calls, and context switches. The loop pipeline handles loop instructions, providing zero-overhead loops. The execution unit also maintains the program counter.

General purpose register file (GPR): The GPR provides 16 address registers and 16 data registers.

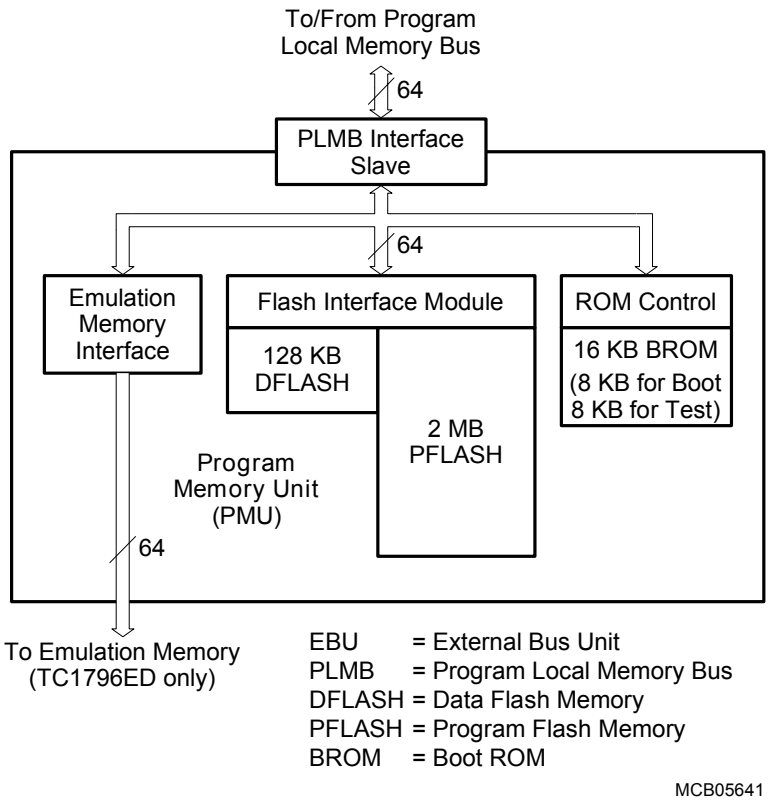

Figure 7.6: Block diagram of the TC1796 program memory unit (PMU). Illustration taken from [Inf07].

CPU slave interface (CPS): The CPS provides accesses to the interrupt service requests registers.

Floating point unit (FPU): The FPU is an optional, partially IEEE-754 compatible component for processing floating-point instructions.

Individual instructions may experience a jitter in execution time, due to pipeline stalls. Figure 7.10 illustrates an example of a pipeline hazard that is resolved by a pipeline stall: In this case, the integer pipeline is processing a multiply-and-accumulate (MAC) instruction, which requires two cycles in the execute stage. At the same time the load/store pipeline is processing a load instruction to the write register of the MAC instruction, which results in a write-after-write hazard.

For conditional branch instructions, the TC1796 uses a simple, static predictor that implements the following rules: Backward and short forward branches (16-bit branches with positive displacement) are predicted taken. Long forward branches are predicted not taken. Table 7.1 summarizes the cycle penalties for each combination of predicted and actual behavior.

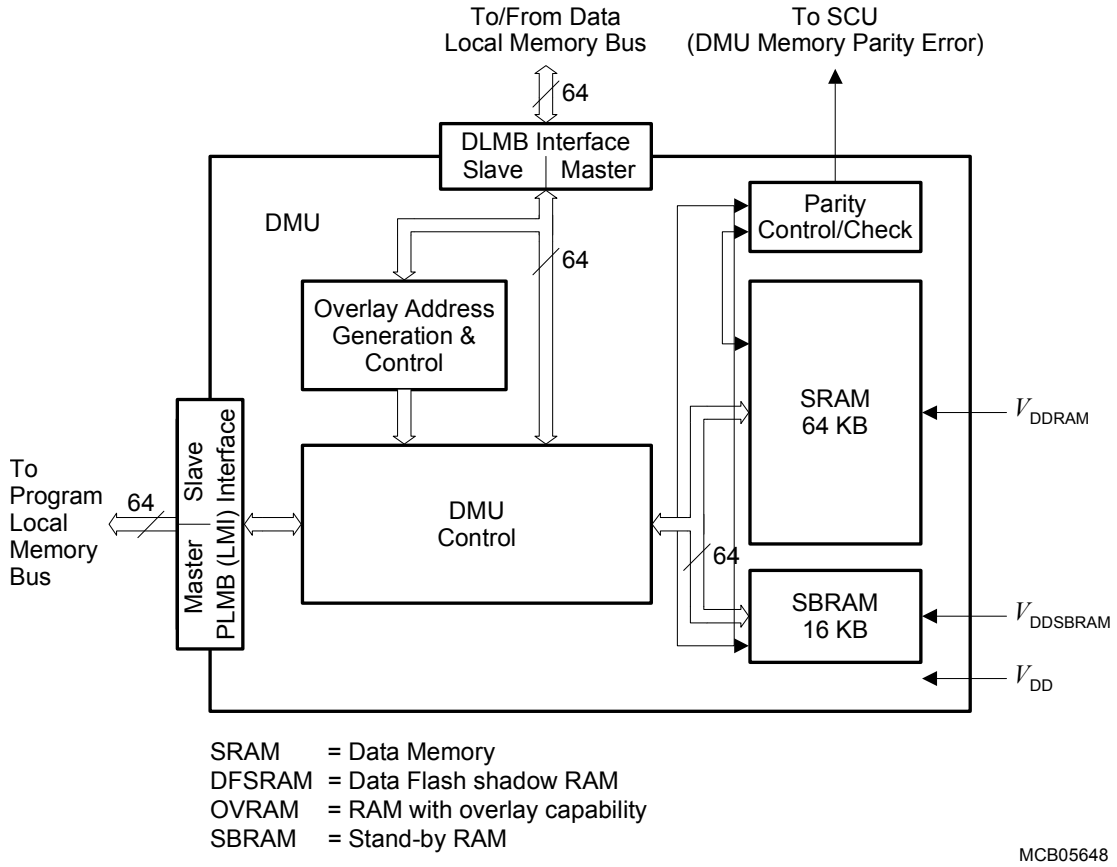

Figure 7.7: Block diagram of the TC1796 data-memory unit (DMU). Illustration taken from [Inf07].

Prediction	Outcome	Penalty (cycles)
not taken	not taken	1
not taken	taken	3
taken	not taken	3
taken	taken	2

Table 7.1: Branch penalties of the TC1796 processor, for all combinations of prediction and actual outcome.

7.2 The OCDS Debugging Interface

The TC1796 provides sophisticated debugging and tracing support that are made available via the *On-Chip Debug Support* (OCDS), a vendor-specific variant of the standardized *Nexus IEEE-ISTO 5001-2003* interface[1].

The OCDS system is split organized in three levels that offer different debugging features:

[1] `http://www.nexus5001.org/`

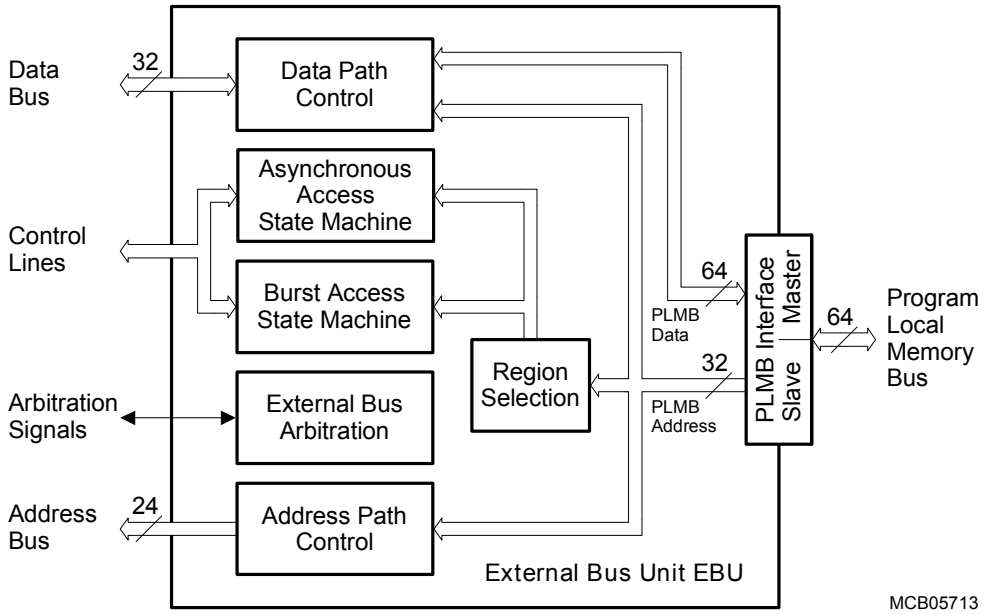

Figure 7.8: Block diagram of the TC1796 external bus unit (EBU). Illustration taken from [Inf05].

OCDS Level 1: This basic debugging level is based on the industry-standard JTAG interface, which is widely used for debugging and development purposes. It offers control over the internal buses, permits the reading and writing of registers and memories, offers conditional breakpoints, and provides features for halting, starting, and single-stepping execution.

OCDS Level 2: This level provides program-tracing support through a 16-bit wide, clocked trace output port. It affords tracing of

- the program flow of the CPU,
- the program flow of the peripheral control processor (PCP),
- the transaction requests of the DMA controller, and
- the information of the DMA controller move engine.

Because the trace output port is only 16 bits wide, only one of the tracing facilities can be enabled at any given instant.

OCDS Level 3: This level provides a multi-core debug solution (MCDS) that is based on a special TC1796 emulation device, which includes the TC1796 product chip and additional on-chip hardware.

The essential feature for our purpose is the CPU program-flow tracing of OCDS Level 2. By using external special-purpose debugging hardware, it becomes possible to reconstruct cycle-accurate timed traces of the code under analysis. Since the tracing interface is non-intrusive, the traces can be collected without a probe effect on the system.

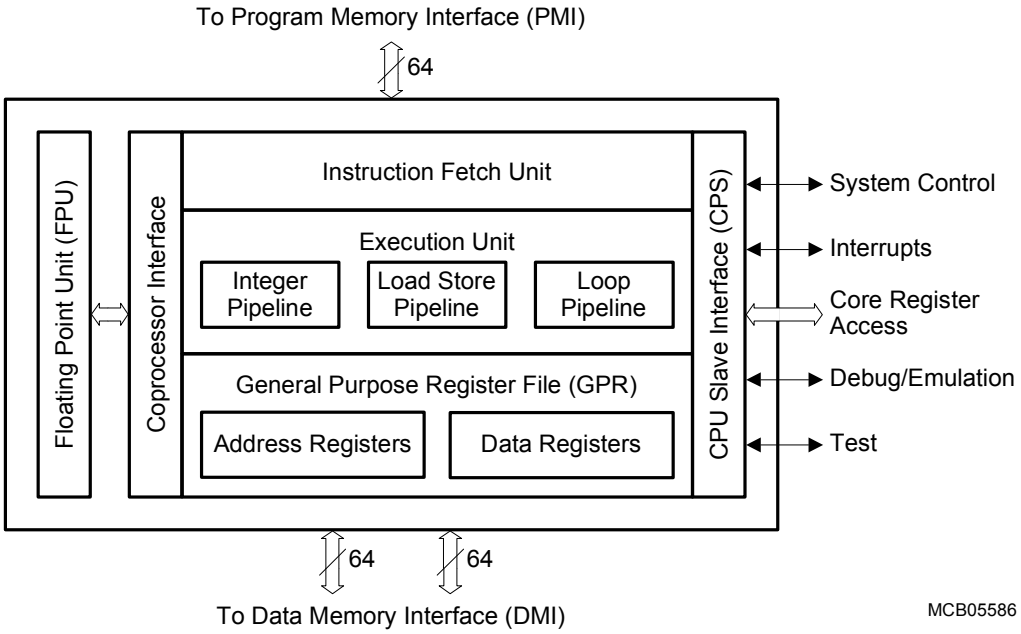

Figure 7.9: Block diagram of the TC1796 central processing unit (CPU), indicating the superscalar pipeline design. Illustration taken from [Inf05].

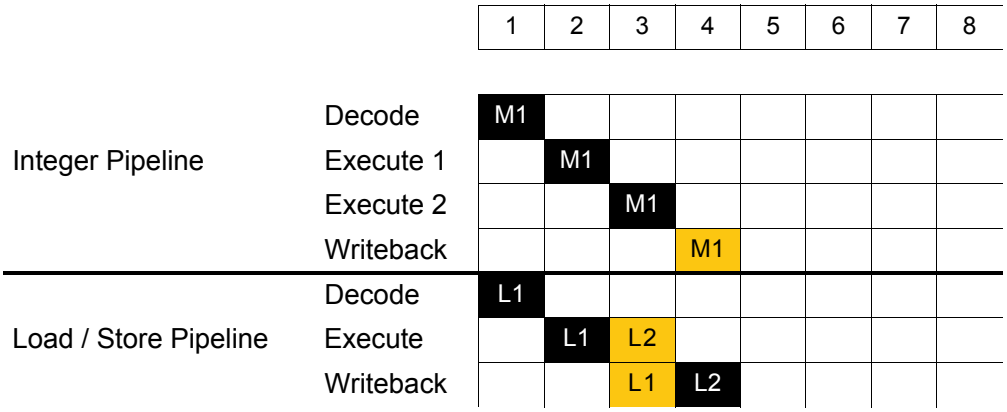

Figure 7.10: Example of a pipeline hazard in the TC1796 CPU. Illustration taken from [Inf00].

In our setup, we used a *Lauterbach PowerTrace* device [Lau], which was equipped with 512MiB of trace memory. This setup allowed us to capture complete end-to-end traces. Figure 7.11 shows a photo of the physical setup of the PowerTrace device and the used evaluation board.

For large programs, the trace memory might be too small for capturing complete end-to-end traces, such as we require for context-sensitive IPET. Apart from the obvious (but limited) solution of using more trace memory, the special start/stop trigger features of the OCDS Level 2 tracing facility would allow the implementation of techniques like trace automata [SM10], in order to capture partial traces. A complete trace could

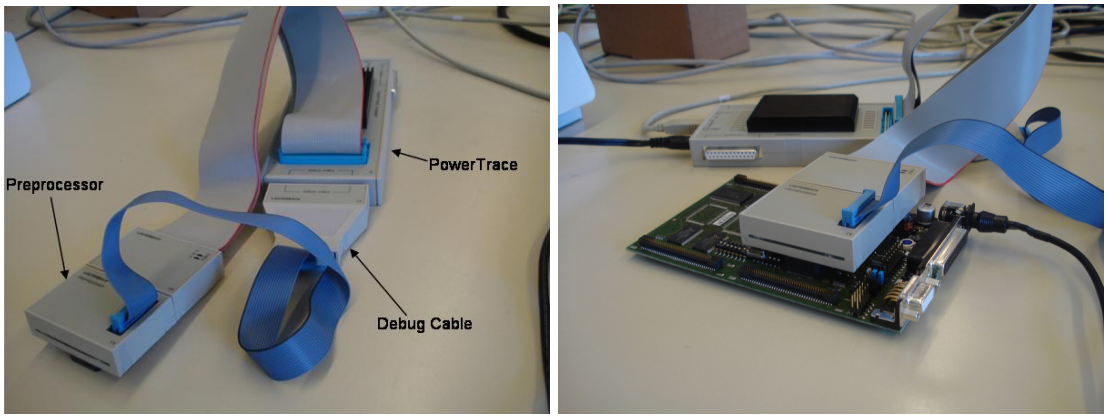

Figure 7.11: Parts of a Lauterbach PowerTrace device (left picture), and connection to an Infineon TriBoard TC179X (right picture).

subsequently be reconstructed from the partial traces. Since we never ran out of tracing memory in our experiments, we did not try to implement this approach.

7.3 The TriBoard TC179X Evaluation Board

The TriBoard is equipped with 4MB of Burst Flash memory and 1 MB of asynchronous SRAM, which are both connected to the processing core via the External Bus Unit of the processor, and these are the only devices that are connected to the EBU (cf. Figure 7.12). The *Clock Generation Unit*, which is controlled by an external crystal oscillator, produces a clock signal f_{OSC} at 20MHz. The CPU clock runs at 150MHz, and the system clock at 75MHz. More details can be found in the board manual [Inf05].

7.4 The Benchmarks

For the evaluation of the FORTAS tool, we used benchmarks from four different benchmark suites:

Industry Study (IS): This benchmark suite was derived from code provided by one of our industrial partners. The code implements an engine controller.

Mälardalen WCET Benchmark Suite (MD): The Mälardalen WCET Benchmark Suite [GBEL10] is a collection of benchmark programs that is specifically designed for the evaluation of WCET analysis tools. It consists of 36 individual programs that were collected from different research groups and tool vendors. We used two of the benchmarks: *bs*, an implementation of binary search over an array of 15 integer elements, and *bsort100*, an implementation of bubble sort over an array of 100 integer elements. For the latter benchmark, input-data generation turned out to be too expensive. We therefore modified the benchmark to use an array of only 10 integer elements.

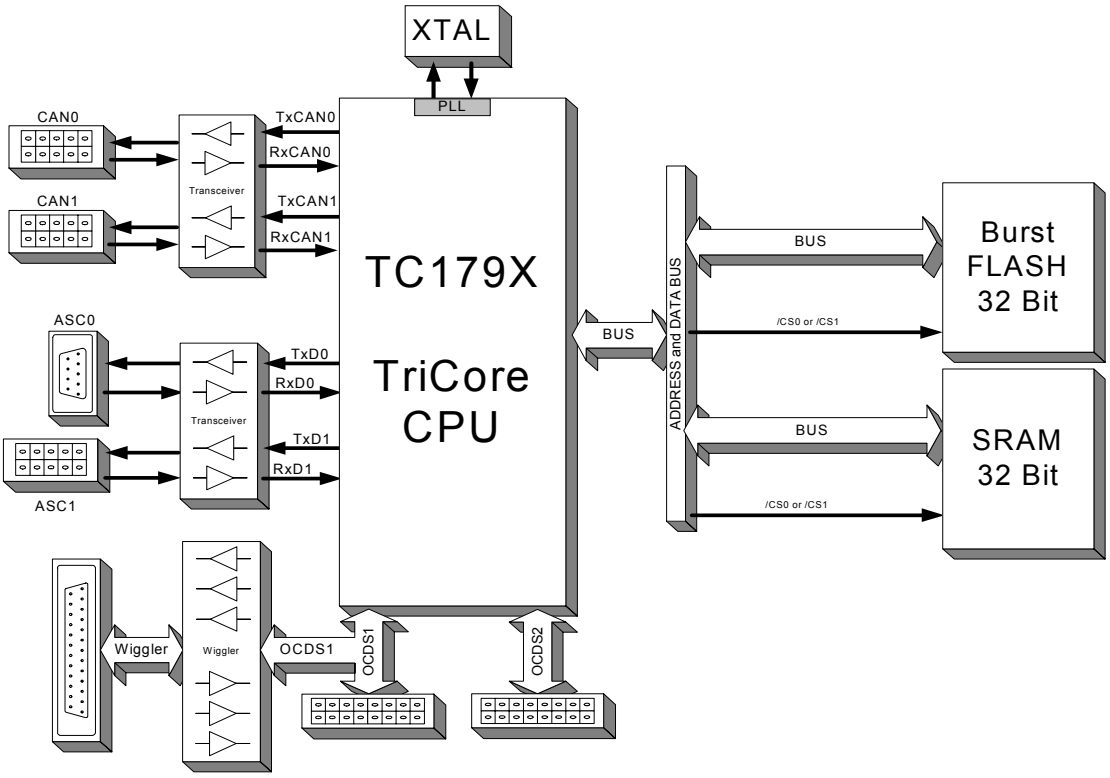

Figure 7.12: Block schematics of the TriBoard-T179X, indicating the connection to external memory, as well as the OCDS debugging interface. Illustration taken from [Inf05].

PapaBench (PB): The PapaBench benchmark originate from the UAV software that had been developed within the Paparazzi project [pap12]. An original benchmark was presented by Nemer et. al. [NCS+06]. Later, a collection of analysis problems was derived from the original benchmark, for use within the WCET Tool Challenge 2011 [vHHL+11]. Since the source code of the benchmark required considerable porting effort to make it work with our prototype analysis tool-chain, we decided to focus on four analysis problems, which we selected arbitrarily out of the full list of problems. The problems we chose were *A1*, *A2*, *F1*, and *F2*.

Java Optimized Processor Benchmark Suite (JOP): The Java optimized processor benchmark suite is a collection of programs that are used for evaluating the Java Optimized Processor (JOP) [Sch09]. We used the central control function of a C port of the *lift control* benchmark.

7.5 Experiments

To evaluate the FORTAS tool, we performed WCET analysis runs of the benchmarks described in Section 7.4. We used two different memory setups:

Internal memory setup: We placed the executable program code in the PMI's scratchpad RAM (SPRAM), and the program data in the DMI's local data RAM

(LDRAM).

External memory setup: We placed the executable program code and the program data in the external SRAM, and enabled the ICACHE.

We chose these two setups, because they represent the extreme cases with respect to temporal jitter: Accesses to the internal memory of the PMI (ICACHE and SPRAM) have a constant penalty of one cycle. There is no source of jitter—such accesses are fully time predictable. The same holds true for accesses to the internal memory of the DMI (DPRAM and LDRAM). On the other hand, accesses to external memory may experience a high access time jitter. Sources of jitter can be found in: possible cache misses, if instruction caching is used; mixed single/block transfers over the PLMB/DLMB; possible occupation of the PLMB/DLMB by another bus master, like the PMU/DMU or the LMI; occupation of the external memory-bus by another bus master; jitter in DRAM accesses.

The two extreme cases for temporal predictability are therefore, on the one hand, the use of the separate internal PMI and DMI memories, and, on the other hand, the shared use of external memory for both, program and data, with instruction caching.

For the internal memory setup, we also performed analyses of the benchmarks using a static WCET analysis tool. For this purpose, we chose the widely used industrial-strength tool aiT [TSH+03].

It was not possible to obtain similar comparative data for the external memory setup, because aiT does not support such a complex configuration. Whereas it is somewhat unsatisfying not to have comparative data, this circumstance does, in fact, a central advantage of MBTA: Here we have an example of a situation where MBTA is able to readily provide WCET estimates, whereas static analysis is not supported—maybe because the hardware is relatively complex, or just new on the market.

Our hypothesis was that the external memory setup could provide evidence of the effectiveness of the strategies for reducing pessimism and optimism that we had built into the FORTAS tool. On the other hand, we expected the results for the internal memory setup to provide evidence that the analysis results of the FORTAS tool are comparable in quality to those of static WCET analysis tools like aiT.

We performed the analyses on an Intel Core2 Quad Q9450 CPU running at 2.66GHz with 8GiB of DRAM. For each benchmark, we generated at least 100,000 timed traces, using FROO as primary input data generator. At each instant in time when at least 1,000 new traces had become available, the timing model was automatically rebuilt, and an intermediate WCET estimate was obtained using context-sensitive IPET. In parallel, we used FShell [HSTV08, HTVS10, HSTV11] to generate additional input data for the individual contexts generated by context-sensitive IPET.

7.6 Results

Table 7.2 summarizes the analysis results for the internal memory setup. For each benchmark, we list the end-to-end MOET, which is our greatest lower bound of the actual WCET, the WCET estimate produced by the FORTAS tool using context-sensitive IPET, the WCET estimate produced by the FORTAS tool using standard IPET, the WCET bound calculated by aiT, the quotient between the WCET estimates produced

by the FORTAS tool using the two different IPET variants, the number of contexts that were produced by context-sensitive IPET when calculating the final WCET estimate, and the number of CFG nodes.

Table 7.2: Comparison of WCET estimates for internal memory setup.

Benchmark	MOET	FORTAS		AIT	sens.	Con-	No-
		sensitive	standard		/std.	texts	des
is-engine_control_cs1-AktuatorMo.	11.99μs	21.13μs	21.37μs	19.91μs	98%	937	398
jop-lift_control-ctrl_loop	7.39μs	10.45μs	11.00μs	9.91μs	95%	240	119
md-binary_search-binary_search	1.68μs	2.23μs	2.24μs	2.10μs	100%	15	14
md-bsort10-BubbleSort	29.34μs	40.39μs	42.44μs	37.60μs	95%	28	15
pb-a1-course_pid_run	0.91μs	1.35μs	1.41μs	1.89μs	96%	26	17
pb-a1-course_run	1.28μs	1.83μs	1.89μs	2.33μs	97%	12	10
pb-a2-atan2	0.85μs	1.11μs	1.12μs	1.67μs	99%	25	15
pb-a2-compute_dist2_to_home	0.41μs	0.56μs	0.56μs	0.97μs	100%	6	6
pb-a2-nav_home	8.17μs	11.48μs	12.13μs	16.56μs	73%	27	21
pb-a2-navigation_update	8.23μs	11.78μs	12.43μs	16.75μs	100%	4	5
pb-a2-sin	1.77μs	2.69μs	2.71μs	3.88μs	99%	38	29
pb-f1-check_mega128_values_task	2.81μs	3.52μs	3.63μs	4.18μs	97%	16	11
pb-f1-servo_set	2.21μs	2.69μs	2.76μs	3.57μs	97%	81	43
pb-f2-vector_10	1.07μs	1.30μs	1.32μs	1.14μs	98%	22	18

The latter two number should give the reader a rough idea about the quality of the respective analysis problem: the number of contexts gives an indication about how often suitable execution scenarios occur, whereas the number of CFG nodes provides a rough estimate of the size of the analysis problem.

We can see that our observed end-to-end MOET is a consistent lower bound of the respective WCET estimate. This is, of course, what is reasonably expected of an MBTA tool. Moreover, we also expect the end-to-end MOET to be a lower bound of the WCET estimates produced by static WCET analysis tools like AIT. As such tools are expected to produce an upper WCET bound, a failure to exceed the MOET would indicate a severe bug in the tool. However, this was not the case during our measurement.

When we compare the WCET estimates obtained by the FORTAS tool to the WCET estimates produced by AIT, we see that the FORTAS tool returns a higher value for some benchmarks, whereas AIT returns a higher value for some others. Since we have high confidence that the estimates produced by aiT are upper bounds of the actual WCET, we attribute the higher values returned by the FORTAS tool to pessimistic estimate calculation. Nevertheless, it is important to keep in mind that even static WCET analysis tools, like any other piece of software, may contain bugs that may result in erroneous WCET estimates.

The case where the FORTAS tool produces lower WCET estimates than AIT is more difficult to interpret. Without knowledge of the actual WCET of the corresponding benchmark, it is not possible to decide whether or not the FORTAS tool is underestimating the WCET, and how close the estimate is. The distance between the results of both tools can be attributed to the combined effect of optimism and pessimism in the analysis performed by the FORTAS tool, as well as to pessimism in AIT. However, there is no way of deciding to which degree each of these effects contribute to the deviation. Moreover, the magnitude of the deviation does not yield any clue about the precision of the analyses performed by the FORTAS tool and by AIT.

Comparing the WCET estimates produced by the FORTAS tool using context-

sensitive IPET to those produced by the FORTAS tool using standard IPET, we see that the former results are closer to the MOET than the latter. This meets our expectation, since a reduction of pessimism is expected to result in lower WCET estimates, unless there is a comparable reduction of optimism, occurring at the same time. The only parameter that we changed between the two experiments was the used IPET method, whereas we left FROO working in both cases. We therefore have reason to believe that the relative reduction of the WCET estimate observed when switching from standard to context-sensitive IPET might serve as a rough indicator of the achieved reduction in pessimism, illustrating the effectiveness of context-sensitive IPET.

Table 7.3 summarizes the analysis results for the external memory setup. As mentioned before, aiT does not support this more complex configuration, so we can only provide results for the two analysis variants of the FORTAS tool.

When we consider these results, two things become apparent:

Firstly, all MOETs, as well as both WCET estimates are considerably higher than with the internal memory setup. This is not surprising, as dynamic RAM has typically a much higher access latency than static RAM. Moreover, the data path to external memory is much longer than the data path to the SPRAM/LDRAM: The data path to external memory includes, in series, the PMI/DMI, the PLMB/DLMB, the PMU/DMU, the LMI (in case of program data), the EBU, the external memory-bus, and the memory controller of the external memory. On the other hand, the SPRAM/LDRAM is located in the PMI/DMI, which connect directly to the CPU.

Secondly, the distance between the WCET estimates obtained with the FORTAS tool using context-sensitive IPET and the WCET estimates obtained with the FORTAS tool using standard IPET is much larger for the external memory setup than for the internal memory setup—in relative and absolute measures. This corresponds to our expectation, since the data path to external memory contains many sources of temporal jitter, whereas the data path to internal memory is virtually free of jitter.

Table 7.3: Comparison of WCET estimates for external memory setup.

Benchmark	MOET	FORTAS		sens.	Con-	CFG
		sensitive	standard	/std.	texts	Nodes
is-engine_control_cs1-AktuatorMo.	133.45μs	161.09μs	166.25μs	97%	813	398
jop-lift_control-ctrl_loop	64.95μs	73.59μs	87.29μs	84%	319	119
md-binary_search-binary_search	17.76μs	18.32μs	22.61μs	81%	17	14
md-bsort10-BubbleSort	287.89μs	316.23μs	392.93μs	80%	32	15
pb-a1-course_pid_run	8.78μs	9.71μs	10.83μs	90%	25	17
pb-a1-course_run	11.45μs	13.33μs	14.45μs	92%	12	10
pb-a2-atan2	8.29μs	8.58μs	8.66μs	99%	24	15
pb-a2-compute_dist2_to_home	4.02μs	4.16μs	4.16μs	100%	6	6
pb-a2-fly_to_xy	9.27μs	11.87μs	11.95μs	99%	5	5
pb-a2-nav_home	62.21μs	76.18μs	85.81μs	89%	34	21
pb-a2-navigation_update	63.67μs	77.79μs	87.41μs	89%	4	5
pb-a2-sin	15.30μs	17.74μs	19.61μs	90%	46	29
pb-f1-check_mega128_values_task	22.00μs	24.07μs	26.61μs	90%	18	11
pb-f1-servo_set	17.69μs	18.74μs	21.28μs	88%	64	43
pb-f2-vector_10	8.08μs	8.34μs	8.59μs	97%	21	18

To evaluate the effectiveness of the iterative refinement procedure of the FORTAS approach, we have produced plots of the evolution of WCET estimates over the 100 refinement iterations that took place for each benchmark. Figures 7.15 (internal memory setup) and 7.16 (external memory setup) indicate that the preliminary WCET estimates

converge quickly[2], such that useful estimates can be already be obtained after just a few iterations.

A closer inspection of the shape of the curves for the IPET methods provides some more insight into the operation of the FORTAS tool: We can see that, over iteration time, there is mostly a clear upward trend. We attribute this increase of the WCET estimate to the reduction of optimism that is achieved by FROO. The more iterations are performed, the better the coverage of temporal behavior becomes. It can also be seen that both, context-sensitive and standard IPET seem to profit from the decrease in optimism.

Since standard IPET is a monotonically estimate calculation method (cf. Section 4.2), the corresponding curve must be monotonic increasing, which is confirmed by our plots. Context-sensitive IPET is—when used as a standalone method—also a monotonic estimate calculation method. However, the iterative refinement approach in the FORTAS tool can cause estimates to decrease over iteration time, for context-sensitive IPET: Additional timed traces that are collected over time can result in a new choice of contexts. This can, in turn, result in tighter constraints over the execution count of each execution scenario, eventually causing a lower WCET estimate. This effect can be observed in the plot for function `sin` in Figure 7.16.

The WCET estimates obtained by the FORTAS tool using context-sensitive IPET are always below the WCET estimates obtained by the FORTAS tool using standard IPET, which confirms the theoretical result of Theorem 4.1.

As before, we interpret the reduction of the WCET estimate observed when switching from standard to context-sensitive IPET as a rough indicator of the achieved reduction in pessimism. We can see that, for most benchmarks, the distance between the two remains almost constant over iteration time. In some cases, however, the results converge over time. We attribute this effect to an above average increase of the associated WCET estimate of execution scenarios with a previously low associated WCET estimate, due to increased coverage of temporal behavior. In the most extreme case, some previously distinct execution scenarios of a given node may be merged, if their associated execution times have become similar due to newly available data.

For reference, Figures 7.13 and 7.14 provide histograms of the end-to-end MOETs of each benchmark.

From the evaluation, we conclude that our approach can yield reasonable WCET estimates within only a few refinement iterations. These estimates are refined automatically, if the analysis is left running for some more iterations. Furthermore, we have seen the effectiveness of the salient novel features in the FORTAS approach, which are: reduction of pessimism, reduction of optimism, and iterative refinement of the WCET estimate. We have also demonstrated that the WCET estimates that can be obtained using the FORTAS approach are comparable in quality to those obtained from static WCET analysis, by the example of the widely used static WCET analysis tool aiT.

7.7 Chapter Summary

We have presented an experimental evaluation of the FORTAS approach. As target platform, we have chosen a TriBoard TC179X evaluation board, equipped with a Tri-

[2]Note that the horizontal axis of the plots has a logarithmic scale.

Core TC1796 microprocessor. We have chosen this platform, because the TC1796 is a widely used, modern embedded microprocessor. Moreover, since the TC1796 features a fairly complex architecture with many modern, performance-enhancing features, it is an interesting candidate for MBTA, as even widely used static WCET analysis tools like aiT support only a subset of the processor's features. On the other hand, it is easy to achieve support of the TC1796 for MBTA through the implementation of a suitable measurement driver (cf. Section 3.15).

The results of our evaluation show that our approach can yield reasonable WCET estimates. Furthermore, we have demonstrated the effectiveness of the salient novel features in the FORTAS approach, which are: reduction of pessimism, reduction of optimism, and iterative refinement of the WCET estimate. We have also demonstrated that the WCET estimates that can be obtained with the FORTAS approach are comparable in quality to those obtained from static WCET analysis, by the example of the widely used static WCET analysis tool aiT.

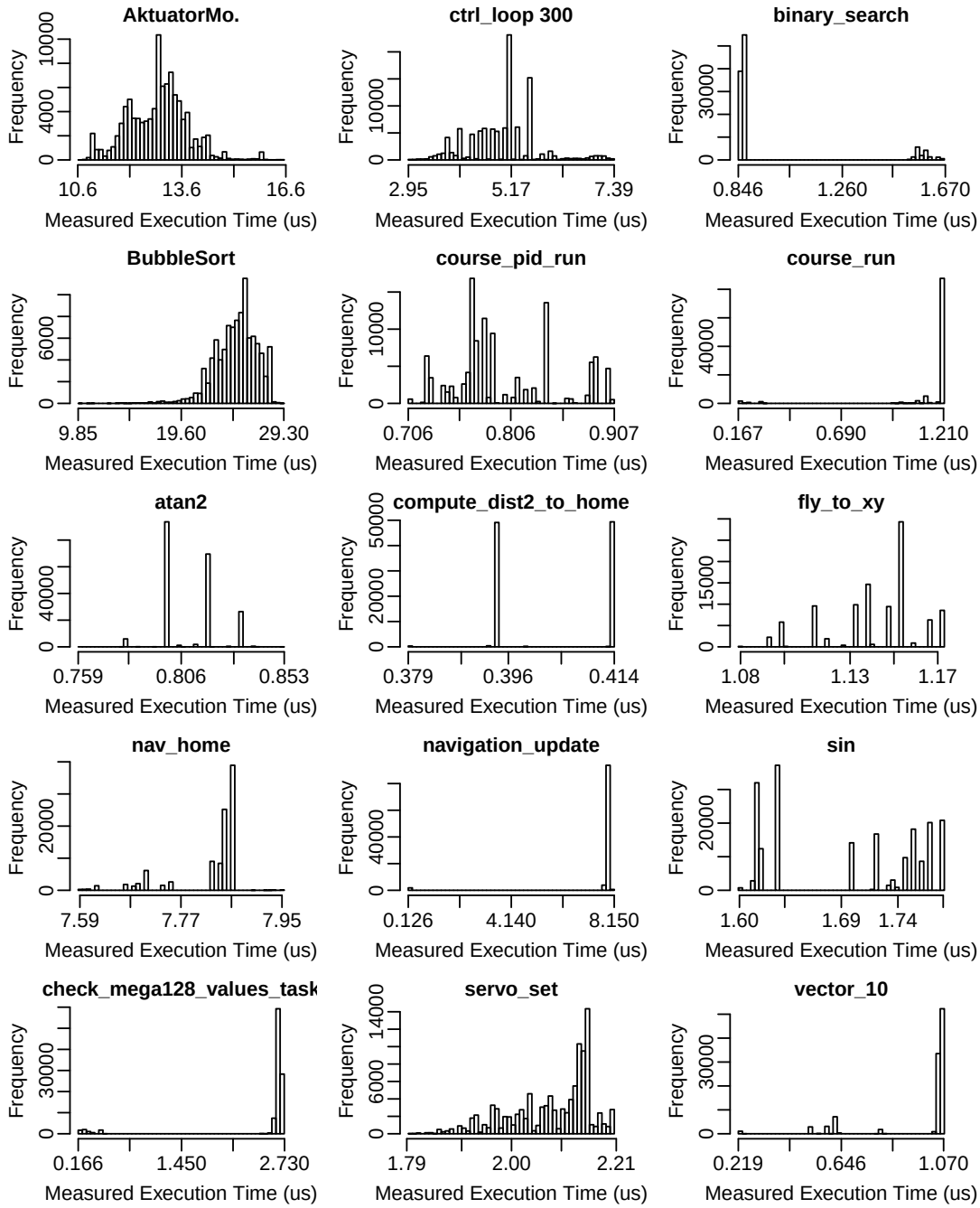

Figure 7.13: Histogram of measured end-to-end execution times for internal memory setup.

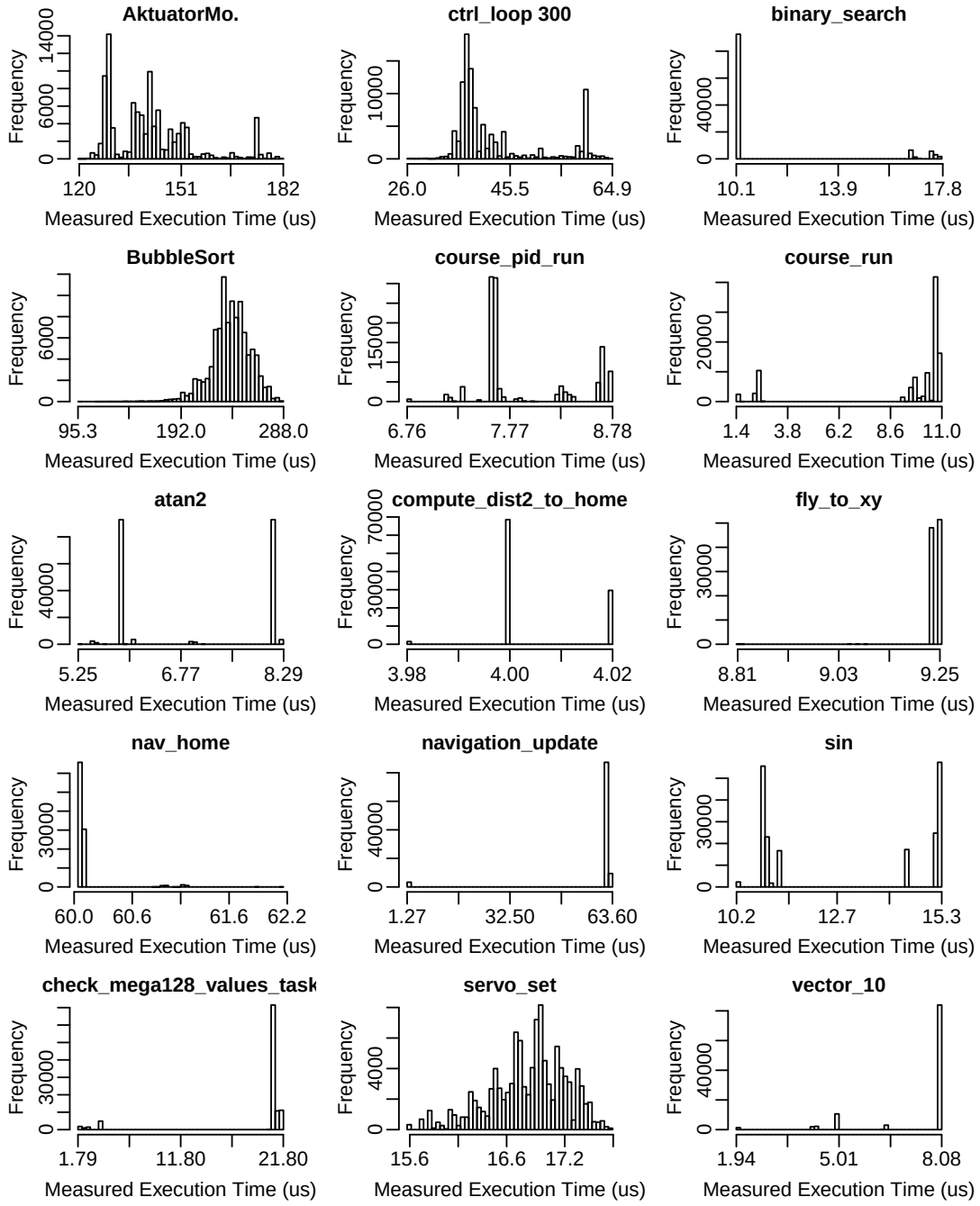

Figure 7.14: Histogram of measured end-to-end execution times for external memory setup.

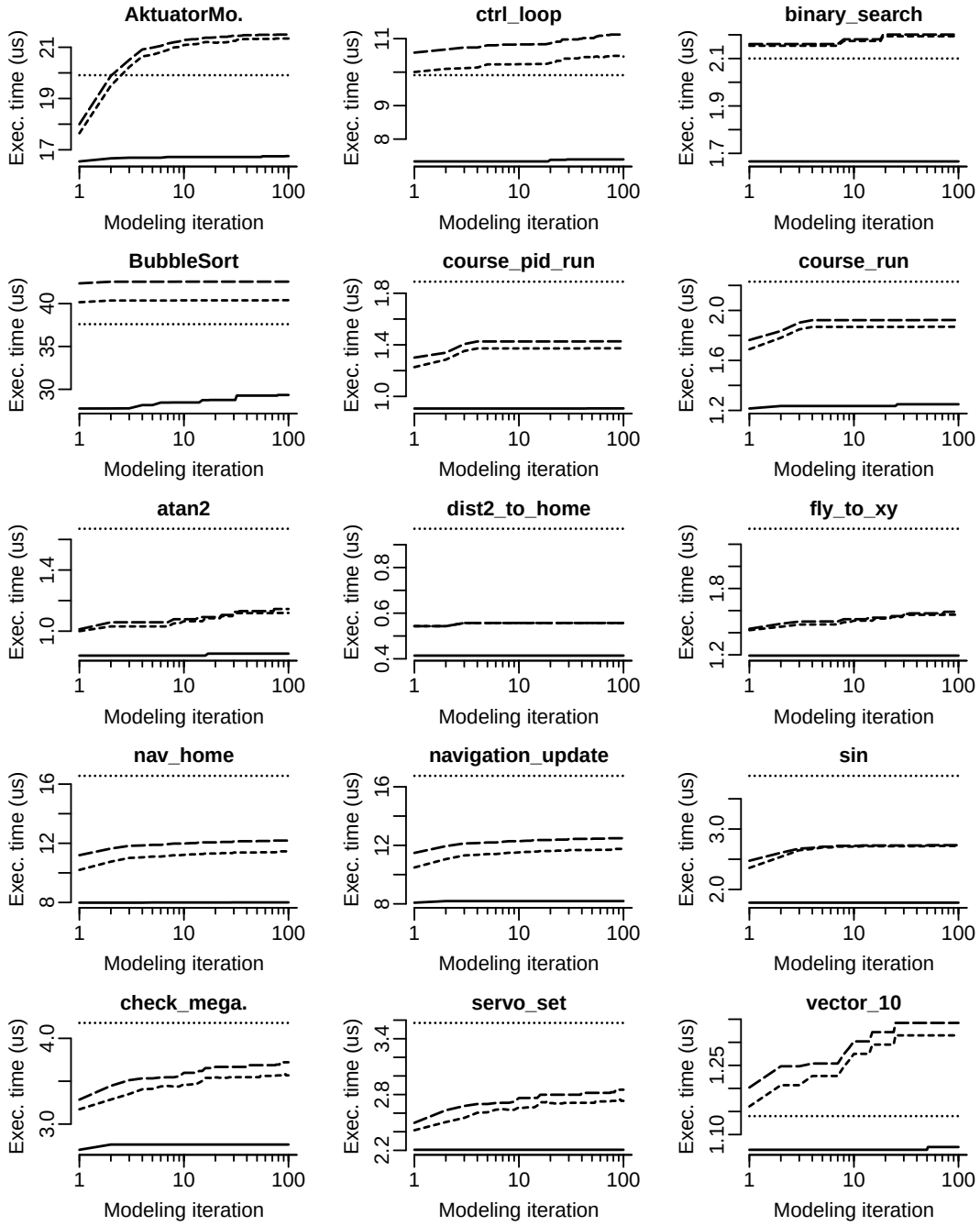

Figure 7.15: Convergence of WCET estimates during model refinement for internal memory setup: end-to-end MOET (solid line), WCET estimate obtained with aiT (dotted line), WCET estimate for MBTA with standard IPET (long-dashed line), and WCET estimate for MBTA with context-sensitive IPET (short-dashed line).

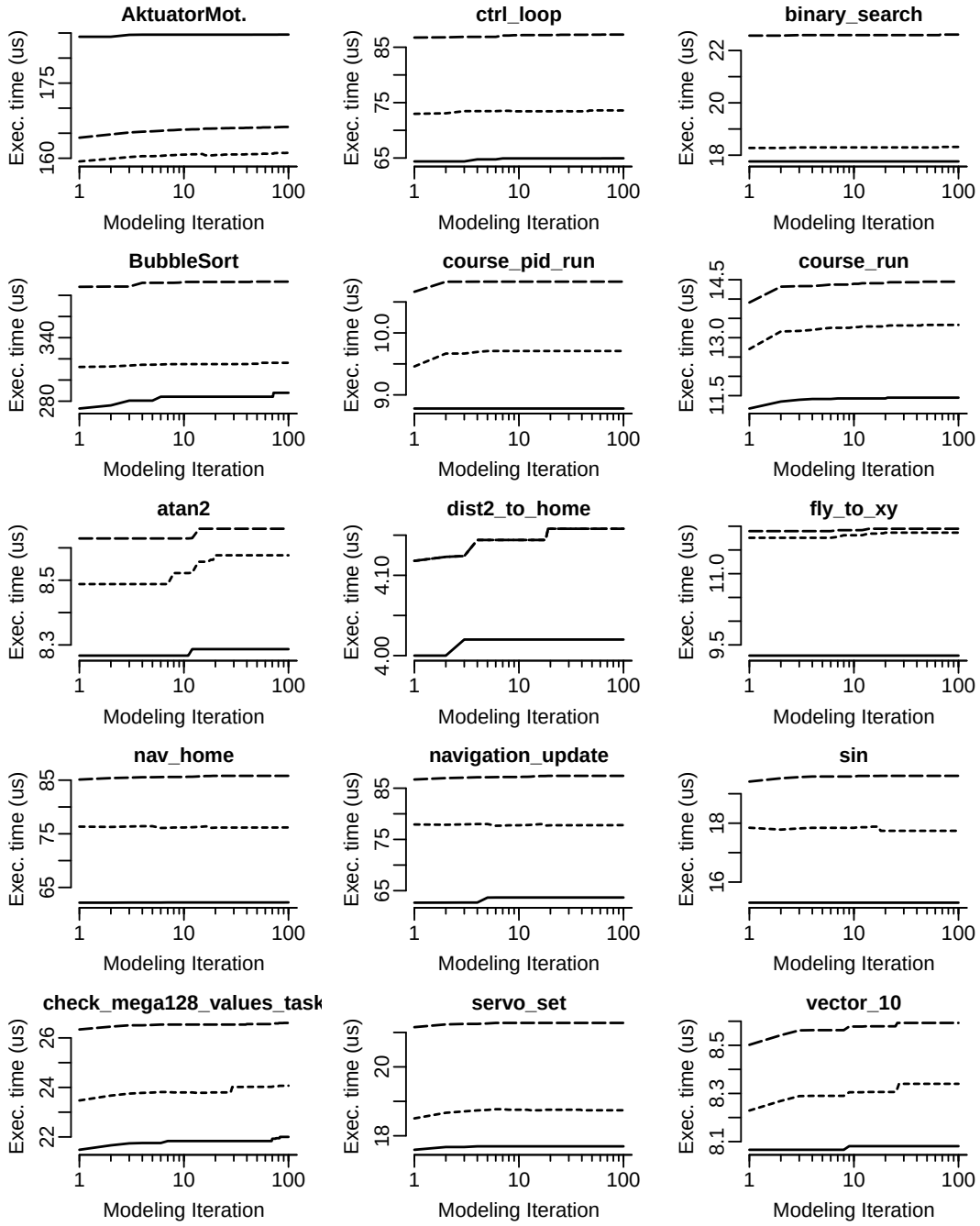

Figure 7.16: Convergence of WCET estimates during model refinement for external memory setup: end-to-end MOET (solid line), WCET estimate for MBTA with standard IPET (long-dashed line), and WCET estimate for MBTA with context-sensitive IPET (short-dashed line).

CHAPTER 8

Conclusion and Outlook

This chapter concludes this thesis, summarizing the main aspects and contributions, and gives an outlook on possible future research.

8.1 Conclusion

Within this thesis, we have presented the current state of our research in high-precision MBTA. We have demonstrated that high-precision MBTA is possible, through the application of smart strategies to reduce the inherent pessimism and optimism of MBTA.

We have presented context-sensitive IPET, a generic estimate calculation method. The method is an extension of standard IPET. Unlike standard IPET, context-sensitive IPET allows for the specification of different execution scenarios. This can help to significantly reduce the pessimism of MBTA.

Context-sensitive IPET is a generic method. To obtain a concrete method, context-sensitive IPET must be instantiated with a concrete notion of an *execution scenario*. We have presented such an instantiation, which is based on the notion of a *context*, where the notion of a context captures a specific control flow within the given program of interest. We have also presented an algorithm that produces suitable sets of contexts that meet the requirements of context-sensitive IPET. The algorithm constructs contexts based on measured execution times in such a way that the application of context-sensitive IPET can effectively reduce pessimism.

Next, we have presented FROO, an input-data generation technique that seeks to reduce the optimism of MBTA through an increase of the measurement coverage of temporal behavior. The method techniques from evolutionary computing with techniques from formal verification, and tries to achieve its goal through a joint increase of the MOET over all program parts.

We then have presented the FORTAS approach for high-precision MBTA. The approach that integrates our instantiation of context-sensitive IPET with FROO. Our implementation follows requirements that had previously been collected from industry. Distinctive features that set the FORTAS approach apart from other MBTA approaches are: tracing measurement, context-sensitive estimate calculation, and adaptive input-data generation.

Moreover, the FORTAS approach supersedes the traditional linear workflow of MBTA with an iterative refinement process: Once a program of interest has been submitted to the analysis process, an iterative refinement loop is started that generates an ongoing stream of test suites that are used for new measurements. The resulting stream of new timed traces is buffered, and, periodically, a new estimate calculation is performed. Adaptive input-data generation makes use of feedback data from measurement and estimate calculation to guide the generation of new test suites. Over time, this ongoing process yields more and closer intermediate WCET estimates.

At any point in time, the user is able to inspect the current WCET estimate. However, there is no clearly defined end of the analysis process. The user may let the process continue until he feels comfortable with the achieved precision, or he might specify an exit condition for the process, like some upper bound on resource usage, or an upper bound on the number of refinement steps that should be performed.

Lastly, we have presented an experimental evaluation of our experimental FORTAS tool. The results of this evaluation show that our approach can yield reasonable WCET estimates. Furthermore, it demonstrates the effectiveness of the salient novel features in the FORTAS approach. Moreover, it demonstrates that the WCET estimates that can be obtained with the FORTAS approach compare to those obtained from static WCET analysis, by the example of the widely used static WCET analysis tool aiT.

In conclusion, we have achieved our goal of demonstrating the feasibility of high-precision MBTA.

Generally speaking, MBTA is a useful approach for WCET estimation that complements static WCET analysis in a number of important use cases, which include

- the development of soft real-time systems,
- the development of mixed-criticality systems,
- the testing and evaluation of static WCET tools,
- the cross-checking of analysis results,
- the development of gracefully-degrading real-time systems, and
- design space exploration.

We believe that our research on an approach for MBTA that can provide high-precision WCET estimates can help in further strengthening the utility of MBTA as an approach for WCET estimation.

8.2 Outlook

Within this thesis, we have presented the current state of our research in high-precision MBTA. We have demonstrated that high-precision MBTA is possible, through the application of smart strategies to reduce the inherent pessimism and optimism of MBTA. However, what we have done can just be seen as a first step towards industrial-strength MBTA. Our current state of research provides plenty of future research problems that to be investigated thoroughly, in order to help MBTA grow into a mature complementary

WCET approach for WCET analysis. In this section, we suggest some of the possible future research directions:

Handling of Pessimism: In Chapter 4, we have presented context-sensitive IPET as a generic estimate calculation method. Unlike standard IPET, context-sensitive IPET allows for the specification of different execution scenarios for a given program part. We have then presented a concrete instantiation of context-sensitive IPET that relies on flow contexts to distinguish different execution scenarios. What we have presented is, of course, not the only possible way to approach the problem. In particular, we find the following research questions particularly interesting:

- *Could pessimism be further reduced by considering contextual timing information?*
  The instantiation of context-sensitive IPET that we have presented considers the control-flow history of a given program part to distinguish different execution scenarios. As our evaluation has shown, this can already help to reduce pessimism. However, the strategy that we have presented does not make any use of contextual timing information. For example, the execution time of a given program part might correlate with the execution time of another program part that is executed before. In other words, the execution time of the previously executed program part might allow the method to draw conclusions about the computer state at a later point of execution. It would be interesting to see in which ways such information can be used in a meaningful way, and whether using it could lead to a further, significant reduction of pessimism.
- *Could pessimism be further reduced by considering the execution future?*
  The instantiation of context-sensitive IPET that we have presented considers the control-flow history, but it does not consider the future control-flow, i.e., the sequence of program parts that were executed after the given program part of interest. The formal framework that we have presented is ready to make use of such information. However, the algorithm for finding contexts that we have presented considers only the execution history.

  The intuition why considering the future control-flow can help with the reduction of pessimism is simple: For any give set of computer states A, there is a certain set B of control flows that may follow. If we observe a control flow that is not in B, we may, by logical contraposition, conclude that the computer state was not in A. This reasoning, of course, requires that we know the exact relation between states and possible control flows that may follow, which is not the case in MBTA. However, the intention of the previous scheme is to provide an intuition, not a complete solution.

  Extending the algorithm to consider both, the execution history and the execution future should not be too difficult, and it would be very interesting to see, to which degree the use of such information could further reduce pessimism. In particular, it would be exciting to investigate whether there is any qualitative difference between historic and future information, on both, a philosophical and practical level, and how this difference affects the resulting

reduction of pessimism. Besides the future control flow, future timing information should also be considered. This overlaps with the previous research question.

- *How should missing measurements be handled?*
  In the presentation of our instantiation of context-sensitive IPET, we have suggested two different strategies of dealing with missing measurements: A conservative strategy, which assumes by default that missing measurements are due to insufficient code coverage, and a progressive strategy that assumes by default that missing measurements are due to infeasible paths. These are two completely opposing, basic strategies of approaching the same problem. It would be interesting to investigate, if there are another other strategies for handling missing measurements.

  In particular, we envision a smart strategy that combines the benefits of the conservative strategy with the benefits of the progressive strategy. For example, when a case of missing measurements is discovered, a smart strategy might try to further investigate the actual cause for the lack of data. It could direct smart input-data generation to find suitable input data to exhibit the behavior in question, and it could apply static analysis and model checking to test whether the path is indeed infeasible. Depending on the outcome of these actions, the previously missing measurements might become available, the particular behavior might be shown infeasible, or it testing might turn out too expensive. The smart strategy might then make a heuristic decision about how to handle the situation.

- *How should function calls be handled?*
  We have presented a variant of context-sensitive IPET that operates on the CFG of a program, but which does not provide any natural support for function calls. The only way in which the presented version of context-sensitive IPET can support function calls is via code inlining. Whereas we have not performed experiments in this direction, we expect this approach to work well in the case of small functions, small call depths, and few call sites. If the functions that are inlined are too big, if the calls too deeply nested, or if there are too many call sites, then the approach might not scale well.

  Supporting function calls in a context-sensitive way is very promising with respect to achieving a further reduction of pessimism: calls from different call sites, possibly with different control-flow history, may yield very different execution times in the individual program parts of the called function, as well as of the called function as a whole.

- *Would less aggressive context splitting be beneficial?* The algorithm for finding contexts that we have presented in this thesis is very aggressive concerning the separation of execution scenarios based on their temporal behavior: It always tries to produce separate contexts, even if their associated WCET estimates differ only marginally. Keeping similar execution scenarios together in a single context may result in a smaller IPET problem. Moreover, aggressive separation strategies tends to produce many small contexts, whereas a large, coherent context may enable tighter constraints on its execution count.

It would be interesting to investigate the effect on the performance of the used ILP solver, as well as on the closeness of the resulting WCET estimate.

- *What is the complexity of the presented approach?*
  In this thesis, we have not provided any performance analysis of the generic context-sensitive IPET or our concrete instantiation thereof. Whereas our method performed very well during evaluation, it is unclear how well it scales with program size, or how different program structures and different temporal behavior of the software on a particular target platform can affect the method's performance. A formal complexity analysis would be very useful towards this end. Moreover, it would be interesting to collect real-world performance data over a representative set of benchmarks.

  An experimental performance evaluation of our prototypical FORTAS tool is, however, not straightforward. Our implementation of context-sensitive IPET interacts tightly with other parts of the analysis framework, like, for example, input-data generation, and it is not clear to which degree the resource consumption of other modules should be accounted. Moreover, the high parallelism of our implementation, as well as the cyclic dependency of services within the refinement loop complicates a clean performance evaluation.
- *How do different ILP solvers perform on context-sensitive IPET?*
  In our prototypical implementation of the FORTAS approach, we are using the free LP solver LPSOLVE to solve ILP problems. It would be interesting to investigate, whether different solvers can perform significantly better on solving the particular class of problems that are produced by context-sensitive IPET.

Handling of Optimism: In Chapter 5, we have presented FROO as strategy for reducing the inherent optimism of MBTA[1] . As is the case with our solution for reducing pessimism, there is plenty of room for further investigation into strategies for reducing optimism. The particular research questions that we find interesting are:

- *How should function calls be handled?*
  We have already encountered this question in connection with the handling of pessimism. Like context-sensitive IPET, the version of FROO that we have presented in this thesis operates on the CFG of a program, but does not provide any natural support for function calls. Again, it is possible to support function calls via inlining, under the known scalability constraints that we mentioned earlier. A particularly interesting research questions concerning the possibility of natively supporting function calls in FROO is, which coverage metrics should be used.
- *How can FROO be integrated more tightly with context-sensitive IPET?*

[1]Notice: The results presented in Chapter 5 have mostly been obtained through the research effort of Sven Bünte, as part of our research collaboration within the FORTAS project. The author of this thesis provides this overview for the sake of completeness, but does not claim any scientific results in that chapter as his own.

In the current version of the FORTAS tool, the integration of FROO with context-sensitive IPET is rather loose, and work only in one direction, in the sense that context-sensitive IPET profits from any increase in coverage of temporal behavior that FROO can achieve.

There is, however, ample room for much greater synergy between the two methods. A first step towards a tighter integration of the two methods could be achieved by teaching FROO the notion of a context. If FROO could operate on individual contexts, rather than on the whole CFG, it could focus on reducing optimism in places where such a reduction is really needed.

An example of an improvement that could be achieved this way is a reduction of bogus contexts. A bogus context is a context that appears to have a low WCET, but actually suffers from high optimism, due to insufficient coverage. To obtain close WCET estimates, it is important to reduce the number of such contexts, and one way to achieve this would be to direct FROO to increase their optimism.

FORTAS: In Chapter 6, we have presented the FORTAS approach for high-precision MBTA. The approach integrates our strategies for reducing pessimism and optimism. The most interesting open research question is, how the individual processes within the FORTAS approach can be integrated more tightly. We have already mentioned some ideas on how a tighter integration of FROO and our instantiation of context-sensitive IPET could be achieved. Here we present some more research directions concerning integration, as well as engineering issues:

- *How can the* FORTAS *approach perform a WCET analysis on big programs?*
  Our evaluation of the FORTAS approach has demonstrated that our approach works well for rather small programs. However, to turn the FORTAS approach into an industrial-strength approach, strategies must be developed to handle larger programs.

  A first step would, of course, be the identification of potential performance bottle necks. A formal complexity analysis of our algorithms would help here. Moreover, it would be interesting to see how the FORTAS tool performs on larger benchmarks.

  The question of scalability cross-cuts through all the central processes in the FORTAS approach: control-flow analysis, input-data generation, measurement, and context-sensitive estimate calculation. It seems that a relaxation of coverage will be unavoidable for larger problems. A central research question is, how to relax coverage in a smart way, i.e., without sacrificing too much precision.
- *How can the* FORTAS *tool automatically infer flow information?*
  Our current implementation of the FORTAS approach requires manual specification of loop iteration constraints. This is very inconvenient for users, who would likely prefer a fully automatic WCET analysis (cf. Section 6.1). There are existing third-party tools that are able to infer such constraints automatically in many common cases (cf. Section 3.8). Integrating such tools into our implementation of the FORTAS approach is merely a matter of engineering effort.

Evaluation: In Chapter 7, we have presented an experimental evaluation of the FORTAS tool. Our evaluation was good enough as a proof of concept, demonstrating that high-precision MBTA is possible, though the integration of smart strategies for reducing pessimism and optimism. However, the presented evaluation is too small to see how the FORTAS approach performs in different settings. For example, we have claimed that MBTA methods like the ones applied in the FORTAS approach are suitable for design space exploration to help the designer in evaluating different choices of target hardware, algorithms, task factorization, etc. However, we have not provided an evaluation of the FORTAS approach with respect to this claim. Here we present some directions in which the current evaluation could be extended:

- *How does the* FORTAS *tool perform for different execution platforms?*
  It would be interesting to see how the FORTAS tool performs for different target platforms. In particular, it would be very useful to have a measurement driver for a very simple execution platform. This would allow a detailed investigation of the operation of the FORTAS tool in a controlled environment. Most importantly, a very simple execution platform would allow the calculation of the exact WCET, for certain programs, i.e., it would be possible to compare the WCET estimates provided by the FORTAS tool to the actual WCET.
- *How does the* FORTAS *tool perform for different benchmarks?*
  Our evaluation provides only a few, small benchmarks. While it serves well as a proof of concept, it does not provide a complete picture of how the FORTAS tool performs on different programs of different size, from different application domains. It would therefore be interesting to perform a thorough evaluation of the FORTAS tool, on a large number of benchmarks, including some more complex ones.

Bibliography

[ABRW91] Neil C. Audsley, Alan Burns, Mike F. Richardson, and Andy J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In Wolfgang A. Halang and Krithi Ramamritham, editors, *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS'91)*, pages 127–132, Pergamon Press, 1991.

[Alt96] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real-Time Systems (EURWRTS'96)*, pages 102–107, L'Aquila , Italy, June 1996.

[Ata03] Pavel Atanassov. *Experimental Assessment of Worst-Case Program Execution Times*. PhD thesis, Technische Universität Wien, 2003.

[Aud93] Neil C. Audsley. Deadline monotonic scheduling theory and application. *Control Engineering Practice*, 1:71–78, 1993.

[BB06] Adam Betts and Guillem Bernat. Tree-based WCET analysis on instrumentation point graphs. In *9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 558–565, Gyeongju, Korea, 2006. IEEE.

[BBD+10] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. In *Proc. 35th International Symposium on on Mathematical Foundations of Computer Science (MFCS'10)*, volume 6281 of *Lecture Notes in Computer Science (LNCS)*, pages 90–101, Brno, Czech Republic, August 2010. Springer.

[BBD11] Sanjoy Baruah, Alan Burns, and Robert I. Davis. Response-time analysis for mixed criticality systems. In *Proc. 32nd IEEE Real-Time Systems Symposium (RTSS'11)*, pages 34–43, Vienna, Austria, November 2011. IEEE.

[BC08] Clément Ballabriga and Hugues Cassé. Improving the WCET computation time by ipet using control flow graph partitioning. In Raimund Kirner, editor, *Proc. 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'08)*, Prague, Czech Republic, July 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[BC09] J. Bhasker and Rakesh Chadha. *Static Timing Analysis for Nanometer Designs*. Springer, http://dl.acm.org/citation.cfm?id=1611282, 2009.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd Real-Time Systems Symposium (RTSS'02)*, pages 279–288, Austin, Texas, USA, December 2002.

[BCP03] Guillem Bernat, Antoine Colin, and Stefan Petters. pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. In *3rd International Workshop on Worst-Case Execution time Analysis (WCET'03)*, pages 21–38, Porto, Portugal, January 2003.

[BdS91] Frédéric Boussinot and Robert de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select: A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, April 1975.

[BHJM07] Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):505–525, 2007.

[BL00] Francisco Barat and Rudy Lauwereins. Reconfigurable instruction set processors: A survey. In *Proc. 11th IEEE International Workshop on Rapid System Prototyping (RSP'00)*. IEEE, 2000.

[BLS10] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, pages 13–22, Stockholm, Sweden, April 2010.

[BMB10] Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 54–63, Brussels, Belgium, 2010. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Bob07] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer, 2007.

[BP82] Phillip Bruce and Sam M. Pederson. *The Software Development Project: Planning and Management*. John Wiley and Sons, 1982.

[BT08] Sven Bünte and Michael Tautschnig. A benchmarking suite for measurement-based WCET analysis tools. In *Proc. International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)*, pages 353–356, Lillehammer, Norway, April 2008.

[BZK11] Sven Bünte, Michael Zolda, and Raimund Kirner. Let's get less optimistic in measurement-based timing analysis. In *6th IEEE International Symposium on Industrial Embedded Systems (SIES'11)*, Los Alamitos, CA, June 2011. IEEE.

[BZTK11] Sven Bünte, Michael Zolda, Michael Tautschnig, and Raimund Kirner. Improving the confidence in measurement-based timing analysis. In *14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11)*, Newport Beach, CA, USA, March 2011. IEEE.

[CB02] Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proc. 14th Euromicro Conference on Real-Time Systems (ECRT'02)*, Vienna, Austria, 2002. IEEE.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. *Ingénieurs de l'Automobile*, 807:36–42, September 2010.

[CGP00] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *Transactions on Information and System Security (TISSEC)*, 12(2):10:1–10:38, December 2008.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, Barcelona, Spain, 2004. Springer.

[Col01] Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer Series in Statistics. Springer, 1st edition, 2001.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, 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 (ECRTS'01)*, volume 0, Delft, The Netherlands, 2001. IEEE.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Thomas Ball and Robert B. Jones, editors, *Proc. 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418, Seattle, WA, USA, August 2006. Springer.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, October 2011.

[Dij70] Edsger Dijkstra. Notes on structured programming. circulated privately, April 1970.

[DM89] Michael Dertouzos and Aloysius K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Trans. Softw. Eng.*, 15(12):1497–1506, December 1989.

[EA99] Edwin Erpenbach and Peter Altenbernd. Worst-case execution times and schedulability analysis of statecharts models. In *Proc. 11th Euromicro Conference on Real-Time Systems(ECRTS'99)*, pages 70 –77, York , UK, 1999.

[EE99] Jakob Engblom and Andreas Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, pages 88–95, Hong Kong, China, December 1999.

[EE00] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pages 163–174, Orlando, Florida, November 2000.

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *3rd International Euro-Par Conference*, volume 1300, pages 1298–1307, Passau, Germany, August 1997. Springer.

[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 *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, Pisa, Italy, July 2007.

[FCWT11] Nathan Fisher, Jian-Jia Chen, Shengquan Wang, and Lothar Thiele. Thermal-aware global real-time scheduling and analysis on multicore systems. *Journal of Systems Architecture - Embedded Systems Design*, 57(5):547–560, 2011.

[FHLS+08] Christian Ferdinand, Reinhold Heckmann, Thierry Le Sergent, Daniel Lopez, and Xavier Fornari. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executable. In *Proc. 4th European Congress on Embedded Real Time Software (ERTS'08)*, Toulouse, France, January 2008.

[FHRF08] Christian Ferdinand, Reinhold Heckmann, Kai Richter, and Patrick Frey. Timing correctness and model-based software development for safety-critical automotive applications - an integrated, tool-supported workflow. In Herbert Klenk, Hubert B. Keller, Erhard Plöderer, and Peter Dencker,

editors, *Automotive - Safety and Security 2008*, Softwaretechnik, Suttgart, Germany, November 2008.

[FHW+08] Christian Ferdinand, Reinhold Heckmann, Hans-Jörg Wolff, Christian Renz, and Reinhard Wilhelm. Towards model-driven development of hard real-time systems. In Manfred Broy, Ingolf Krüger, and Michael Meisinger, editors, *Model-Driven Development of Reliable Automotive Services*, volume 4922, pages 145–160. Springer, 2008.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2-3):163–189, November 1999.

[Fow04] Kim Fowler. Mission-critical and safety-critical development. *IEEE Instrumentation Measurement Magazine*, 7(4):52–59, December 2004.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks - past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis*, Brussels, Belgium, July 2010.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Rio de Janeiro, Brazil, 2006. IEEE.

[God05] *DART: Directed Automated RandomTesting*, Chicago, IL, USA, 2005.

[HAM+99] Christopher Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53 –70, January 1999.

[HBL+95] Yerang Hur, Young Hyun Bae, Sung-Soo Lim, Sung-Kwan Kim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Minsuk Lee, Heonshik Shin, and Chong Sang Kim. Worst case timing analysis of risc processors: R3000/r3010 case study. In *Proc. 16th IEEE Real-Time Systems Symposium (RTSS'95)*, pages 308 –319, Pisa, Italy, December 1995.

[Hig11] Tricore development platform uad2. available for download at corporate website, April 2011. product leaflet.

[HN96] David Harel and Amnon Naamad. The statemate semantics ofstatecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, October 1996.

[How76] William E. Howden. Experiments with a symbolic evaluation system. In *Proc. National Computer Conference and Exposition (AFIPS'76)*, pages 899–908, New York, NY, USA, 1976. ACM.

[HS02] Niklas Holsti and Sami Saarinen. Status of the bound-t WCET tool. In *Proc. 2nd Workshop on Worst-Case Execution Time Analysis (WCET'02)*, Vienna, Austria, June 2002.

[HSRW98] Christopher Healy, Mikael Sjödin, Viresh Rustagi, and David B. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, Denver, CO, USA, June 1998. IEEE Computer Societ.

[HSTV08] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Fshell: Systematic test case generation for dynamic analysis and measurement. In *Proc. 20th International Conference on Computer Aided Verification (CAV'08)*, volume 5123 of *LNCS*, pages 209–213, Princeton, NJ, USA, 2008. Springer.

[HSTV11] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. An introduction to test specification in fql. In Sharon Barner, Daniel Kroening, and Orna Raz, editors, *Proc. Haifa Verification Conference (HVC'10)*, volume 6504 of *Lecture Notes in Computer Science*, pages 9–22, Haifa, Israel, 2011. Springer.

[HTVS10] Andreas Holzer, Michael Tautschnig, Helmut Veith, and Christian Schallhart. How did you specify your test suite? In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 407–416, Antwerp, Belgium, September 2010. ACM.

[HW99] Christopher Healy and David B. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, Vancouver, British Columbia, Canada, 1999. IEEE.

[HWH95] Christopher 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'95)*, pages 288–297, Pisa, Italy, December 1995.

[II03] IEEE-ISTO. The nexus 5001 forum standard for a global embedded processor debug interface. Technical report, IEEE-Industry Standards and Technology Organization (IEEE-ITSO), 445 Hoes Lane, Piscataway, NJ 08854, USA, 2003.

[Inf00] Infineon Technologies AG, St.-Martin-Strasse 53, D-81541 München, Germany. *TriCore(TM) 1 Pipeline Behaviour and Instruction Execution Timing*, January 2000.

[Inf05] Infineon Technologies AG, St.-Martin-Strasse 53, D-81541 München, Germany. *TriBoard TC179X Hardware Manual*, September 2005.

[Inf06] Infineon Technologies AG, St.-Martin-Strasse 53, D-81541 München, Germany. *TriCore 1 32-bit Unified Processor Core*, September 2006.

[Inf07] Infineon Technologies AG, St.-Martin-Strasse 53, D-81541 München, Germany. *TC1796 32-Bit Single-Chip Microcontroller*, July 2007.

[JHCR09] Lei Ju, Bach Khoa Huynh, Samarjit Chakraborty, and Abhik Roychoudhury. Context-sensitive timing analysis of esterel programs. In *Proc. 46th ACM/IEEE Design Automation Conference (DAC '09)*, pages 870 –873, San Francisco, CA, USA, July 2009.

[JHRC09] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. A systematic classification and detection of infeasible paths for accurate WCET analysis of esterel programs. In Wee Kheng Leow, Daniel Racoceanu, Feng Ding, and Hao Li, editors, *Proc. Singaporean-French IPAL Symposium (SinFra'09)*, Singapore, February 2009. World Scientific Publishing.

[JHRC10] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. Timing analysis of esterel programs on general-purpose multiprocessors. In *Proc. 47th ACM/IEEE Design Automation Conference (DAC'10)*, pages 48–51, Anaheim, CA, USA, June 2010.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. *SIGPLAN Notices*, 29(6):171–185, June 1994.

[KB09] Usman Khan and Iain Bate. WCET analysis of modern processors using multi-criteria optimisation. In *Proc. 1st International Symposium on Search Based Software Engineering (SSBSE'09)*, pages 103–112, Cumberland Lodge, Windsor, UK, May 2009.

[KBZ10] Raimund Kirner, Sven Bünte, and Michael Zolda. *Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility*, chapter 5: Measurement-Based Timing Analysis for Reconfigurable Embedded Systems. IGI Global, Hershey, PA, USA, November 2010.

[KGGV07] Milos Krstic, Eckhard Grass, Frank K. Gürkaynak, and Pascal Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *IEEE Design Test of Computers*, 24(5):430–441, 2007.

[KH09] Raimund Kirner and Walter Haas. Automatic calculation of coverage profiles for coverage-based testing. In *Proc. 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Maria Taferl, Austria, October 2009.

[Kir08] Raimund Kirner. *Compiler Support for Timing Analysis of Optimized Code: Precise Timing Analysis of Machine Code with Convenient Annotation of Source Code.* VDM, Saarbrücken, Germany, July 2008.

[Kir09] Raimund Kirner. Towards preserving model coverage and structural code coverage. *EURASIP Journal on Embedded Systems*, pages 6:1–6:16, January 2009.

[KK07a] Albrecht Kadlec and Raimund Kirner. On the difficulty of building a precise timing model for real-time programming. In *Proc. 14. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, pages 99–105, Timmendorfer Strand, Germany, October 2007.

[KK07b] Kendall Kim and Jürgen Kaljuvee. *Electronic and Algorithmic Trading Technology: The Complete Guide*. Academic Press, 2007. 978-0-12-372491-5.

[KKP09] Raimund Kirner, Albrecht Kadlec, and Peter Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *Proc. 21st Euromicro Conference on Real-Time Systems*, Dublin, Ireland, July 2009. IEEE.

[KKP10] Albrecht Kadlec, Raimund Kirner, and Peter Puschner. Avoiding timing anomalies using code transformations. In *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'10)*, pages 123–132, Parador de Carmona, Spain, May 2010.

[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 (ECRTS'02)*, pages 31–40, Vienna, Austria, June 2002. IEEE.

[KLPT00] Raimund Kirner, Roland Lang, Peter Puschner, and Christopher Temple. Integrating WCET analysis into a Matlab/Simulink simulation model. In Arcot Sowmya and J. Park, editors, *Proc. 16th IFAC/IFIP Workshop on Distributed Computer Control Systems (DCCS'00)*, pages 79–84, Sydney, Australia, November 2000. Elsevier.

[KMH96] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 230–240, Brookline, MA , USA, June 1996.

[Kni02] John C. Knight. Safety critical systems: Challenges and directions. In *Proc. 24th International Conference on Software Engineering (ICSE'02)*, pages 547–550, Orlando, Florida, 2002.

[Kog81] Peter M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Corporation, 1981.

[Kop97] Hermann Kopetz. *Real-time systems*. The Kluwer international series in engineering and computer science. Kluwer, Boston, Mass., 1997.

[KP01] Raimund Kirner and Peter Puschner. Transformation of path information for WCET analysis during compilation. In *Proc. 13th Euromicro International Conference on Real-Time Systems (ECRTS'01)*, pages 29–36, Delft, The Netherlands, June 2001.

[KP03] Raimund Kirner and Peter Puschner. Timing analysis of optimised code. In *Proc. 8th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*, pages 100–105, Guadalajara, Mexico, January 2003.

[KP05] Raimund Kirner and Peter Puschner. Classification of code annotations and discussion of compiler support for worst-case execution time analysis. In Reinhard Wilhelm, editor, *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, Palma de Mallorca, Spain, July 2005.

[KPP10a] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1):72–105, 2010.

[KPP10b] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Journal of Real-Time Systems*, 45(1-2):72–105, April 2010.

[KPW04] Raimund Kirner, Peter Puschner, and Ingomar Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th International Workshop on Worst-Case Execution Time Analysis (WCET'04)*, pages 67–70, Catania, Italy, June 2004.

[KWH+09] Daniel Kästner, Reinhard Wilhelm, Reinhold Heckmann, Marc Schlickling, Markus Pister, Marek Jersak, Kai Richter, and Christian Ferdinand. Timing validation of automotive software. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 93–107. Springer, 2009.

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

[KZ11] Raimund Kirner and Michael Zolda. Compiler support for measurement-based timing analysis. In *Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET'11)*, Porto, Portugal, July 2011. OCG.

[Lau] Lauterbach GmbH, Höhenkirchen-Siegertsbrunn, Germany. *PowerTrace*.

[LBJ+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, and Chong Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[LL73] Chang L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, January 1973.

[LM97] Yau-Tsun Steven Li and Sharad 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, December 1997.

[LMW95] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software (rtss'95). In *16th IEEE Real-Time Systems Symposium (RTSS'95)*, pages 298–307, Pisa, Italy, 1995.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254 –263, Washington, DC, USA, December 1996. IEEE.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation Electronic Systems*, 4(3):257–279, July 1999.

[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'98)*, pages 1–15, New York, NY, USA, 1998. Springer.

[LS99] 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, December 1999.

[LSHK08] Thierry Le Sergent, Reinhold Heckmann, and Daniel Kästner. Methodology and benefits of timing verification for safety-critical embedded software. Technical white paper, Esterel Technologies, Élancourt, France, 2008.

[LV01] Edward A. Lee and Pravin Varaiya. *Structure and Interpretation of Signals and Systems*. LeeVaraiya.org, second edition, 2001.

[MMS01] David May, Henk L. Muller, and Nigel P. Smart. Non-deterministic processors. In Vijay Varadharajan and Yi Mu, editors, *Information Security and Privacy*, volume 2119 of *LNCS*, pages 115–129. Springer, 2001.

[MT89] Mihajlo D. Mesarovic and Yasuhiko Takahara. *Abstract Systems Theory*. LNCS. Springer, 1989.

[Muc97] Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NCS+06] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne de Michiel. Papabench: a free real-time benchmark. In Frank Mueller, editor, *Proc. 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'06)*, Dresden, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI).

[NNH99] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[OS97] Greger Ottosson and Mikael Sjödin. Worst case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 47–55, Las Vegas, Nevada, 1997. ACM.

[pap12] Paparazzi: The free autopilot. http://paparazzi.enac.fr/, August 2012.

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[PB02] Peter Puschner and Alan Burns. Writing temporally predictable code. In *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02)*, pages 85–91, San Diego, CA, USA, January 2002.

[Pet03] Stefan Petters. Comparison of trace generation methods for measurement based WCET analysis. In *3rd International Workshop on Worst Case Execution Time Analysis (WCET'03)*, Porto, Portugal, July 2003.

[PK89] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[PPVZ92] Gustav Pospischil, Peter Puschner, Alexander Vrchoticky, and Ralph Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35–44, September 1992.

[PS91] Chang Yun Park and Alan Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24:48–57, 1991.

[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 (EURO-RTS'93)*, pages 224–229, Oulu, Finland, June 1993. IEEE.

[PS97] Peter Puschner and Anton Schedl. Computing maximum task execution time - a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, July 1997.

[Pus98] Peter Puschner. A tool for high-level language analysis of worst-case execution times. In *Proc. 10th Euromicro Workshop on Real-Time Systems (Euro-Rts'98)*, pages 130–137, Berlin, Germany, June 1998.

[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, *Proc. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems (DIPES '02)*, pages 163–172, Montreal, Canada, 2002. Kluwer.

[Pus03] Peter Puschner. The single-path approach towards WCET-analysable software. In *Proc. IEEE International Conference on Industrial Technology (ICIT'03)*, pages 699–704, Hotel Habakuk, Maribor, Slovenia, December 2003.

[Pus05] Peter Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)*, pages 205–210, Sedona, AZ, USA, February 2005.

[RML+94] Byung-Do Rhee, Sang Lyul Min, Sung-Soo Lim, Heonshik Shin, Chong Sang Kim, and Chang Yun Park. Issues of advanced architectural features in the design of a timing tool. In *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS'94)*, pages 59–62, Seattle, WA , USA, May 1994.

[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 *Proc. 6th International Workshop on Intelligent Solutions in Embedded Systems (WISES'08)*, pages 1–7, Regensburg, 2008.

[RWSP07] Bernhard Rieder, Ingomar Wenzel, Klaus Steinhammer, and Peter Puschner. Using a runtime measurement device with measurement-based WCET analysis. In *Proc. International Embedded Systems Symposium (IESS'07)*, pages 15–26, Amsterdam, Netherlands, June 2007.

[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, February 2000.

[Sch09] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. CreateSpace, 2009.

[SEE01] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proc. 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, pages 132–140, Atlanta, Georgia, USA, 2001. ACM.

[SF99] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. *SIGPLAN Notices*, 34(7):35–44, May 1999.

[Sha89] Alan Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7), 1989.

[SKL01] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18(2):20–30, March 2001.

[SM10] Stefan Stattelmann and Florian Martin. On the use of context information for precise measurement-based execution-time estimation. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'10)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–76, Brussels, Belgium, July 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proc. 10th European Software Engineering Conference (ESEC'05)*, pages 263–272, Lisbon, Portugal, 2005.

[STA89] *The STARTS Purchasers' Handbook: Software Tools for Application to Large Real Time Systems.* National Computing Centre Publications, second edition, 1989.

[Sto96] Neil R. Storey. *Safety-Critical Computer Systems.* Addison-Wesley Longman, 1996.

[TAK06] Jason G. Tong, Ian D. L. Anderson, and Mohammed A. S. Khalid. Soft-core processors for embedded systems. In *Proc. International Conference on Microelectronics (ICM '06)*, pages 170–173, Dhahran, Saudi Arabia, December 2006.

[The00] Henrik Theiling. Extracting safe and precise control flow from binaries. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, pages 23–30, Cheju Island, South Korea, December 2000.

[TSH+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. 2003 International Conference on Dependable Systems and Networks (DSN'03)*, volume 0, San Francisco, CA, USA, 2003. IEEE.

[vHHL+11] Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Reinhard Wilhelm, Armelle Bonenfant, Hugues Cassé, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. WCET tool challenge 2011: Report. In *Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'11)*, Porto, Portugal, July 2011. The analysis problems for the WCET Tool Challenge can be found at http://www.mrtc.mdh.se/projects/WCC/2011.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, 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.

[WKRP05] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Seattle, WA, USA, May 2005. IEEE.

[WKRP09] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'08)*, volume 17 of *Communications in Computer and Information Science*, pages 430–444, Porto Sani, Greece, October 2009. Springer.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *Dependable Computing - EDCC 5*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Proc. Design, Automation and Test in Europe (DATE'05)*, volume 1, pages 606–611, Munich, Germany, March 2005.

[XP90] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Trans. Softw. Eng.*, 16(3):360–369, March 1990.

[ZBK09] Michael Zolda, Sven Bünte, and Raimund Kirner. Towards adaptable control flow segmentation for measurement-based execution time analysis. In *17th International Conference on Real-Time and Network Systems (RTNS'09)*, Paris, France, October 2009. INRIA HAL.

[ZBK10] Michael Zolda, Sven Bünte, and Raimund Kirner. Context-sensitivity in ipet for measurement-based timing analysis. In *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10)*, Amirandes, Heraclion, Crete, Greece, October 2010.

[ZBK11] Michael Zolda, Sven Bünte, and Raimund Kirner. Context-sensitive measurement-based worst-case execution time estimation. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, Toyama, Japan, August 2011. IEEE.

[ZD04] Fubo Zhang and Eric H. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, 30(4):231 – 245, April 2004.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proc. 18th International Conference on Static Analysis (SAS'11)*, pages 280–297, Venice, Italy, 2011. Springer.

List of Figures

List of Tables

Image Sources

Hyundai DVD player (Figure 2.1): http://upload.wikimedia.org/wikipedia/commons/8/8f/Hyundai_dvd_player.jpg, licensed under Creative Commons Attribution-Share Alike.

Capsule endoscope (Figure 2.1): http://upload.wikimedia.org/wikipedia/commons/f/f9/CapsuleEndoscope.jpg, in the public domain.

Block II-F GPS satellite (Figure 2.1): http://upload.wikimedia.org/wikipedia/commons/8/8d/GPS_Satellite_NASA_art-iif.jpg, in the public domain.

da Vinci Surgical System (Figure 2.1): http://upload.wikimedia.org/wikipedia/commons/0/0d/Laproscopic_Surgery_Robot.jpg, licensed under Creative Commons Attribution-Share Alike.

Boeing 787 "Dreamliner" (Figure 2.1): http://upload.wikimedia.org/wikipedia/commons/5/50/Boeing_787-8_maiden_flight_overhead_view.jpg, licensed under Creative Commons Attribution-Share Alike.

Industrial laser cutter (Figure 2.1): http://upload.wikimedia.org/wikipedia/commons/c/c4/Industrial_4kW_laser_with_flying_optics_system.jpg, licensed under Creative Commons Attribution-Share Alike.

NASA Mars Science Laboratory "Curiosity" Rover (Figure 2.1): http://photojournal.jpl.nasa.gov/catalog/PIA14156, in the public domain.

NOAA ROV "Hercules" (Figure 2.1): http://upload.wikimedia.org/wikipedia/commons/3/3d/ROV_Hercules_2005.JPG, in the public domain.

General Atomics MQ-1 Predator UAV (Figure 2.1): http://www.af.mil/shared/media/photodb/photos/081131-F-7734Q-001.jpg, in the public domain.

Curriculum Vitae: Dipl.-Ing. Michael Zolda

Personal Information

Date of Birth: 08/02/1979
Citizenship: Austrian

Contact Information

School of Computer Science
University of Hertfordshire
Hatfield, AL10 9AB

Phone: +44 1707 281294
Fax: +44 1707 284303
Email: m.zolda@herts.ac.uk

Main Area of Research

Modeling, design, verification, and testing of *real-time, embedded, and cyber-physical systems*. Earlier contributions in *foundations of knowledge-based systems*.

Academic career

1. **Researcher** in Computer Engineering (from July 2012 to present), University of Hertfordshire, United Kingdom.
2. **Researcher** in Computer Engineering (from May 2007 to March 2011), Vienna University of Technology, Austria.
3. **Teaching Assistant** for Theoretical Computer Science (from March 2004 to June 2005), Department of Computer Languages, Department of Computer Languages, Vienna University of Technology, Austria.
4. **Teaching Assistant** for Logic Programming (from September 2000 to January 2004), Department of Computer Languages, Department of Computer Languages, Vienna University of Technology, Austria.
5. **Teaching Assistant** for Network Services (from March 1999 to June 2000), Department of Foreign Affairs, Vienna University of Technology, Austria.

Education

Dipl.-Ing. in Computer Science (January 2005)
Vienna University of Technology, Austria - graduated with honors
Thesis: *Comparing Different Prenexing Strategies for Quantified Boolean Formulas*
Advisor: Prof. Uwe Egly, Co-Advisor: Dr. Stefan Woltran

Academic Prizes/Recognition

Scholarship for Outstanding Achievements (November 2003)
Faculty for Technical and Natural Sciences, Vienna University of Technology, Austria.

Support Scholarship (May 2003)
Faculty for Technical and Natural Sciences, Vienna University, Austria.

Support Scholarship (June 2002)
Lower Austrian Academy.

Peer Review Activities

Journal Referee: Real-Time System (RTS), 2012, Journal of Systems Architecture (JSA), 2010.

Book Referee: Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility, IGI Global, Hershey, PA, USA, 2011.

Conference Referee: Design, Automation and Test in Europe (DATE), 2011, IFIP Conference on Distributed and Parallel Embedded Systems (DIPES), 2010, Euromicro Conference on Real-Time Systems (ECRTS): 2010, 2011, 2012. International Conference on Software Testing, Verification and Validation (ICST): 2010. International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), 2008. IEEE International Symposium on Object/Component/Service-oriented Real-Time Distributed Computing (ISORC), 2011. International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008, 2009. International Conference on Real-Time and Network Systems (RTNS), 2008, 2011. IEEE Real-Time Systems Symposium (RTSS), 2007, 2008, 2009, 2011. IEEE International Symposium on Industrial Embedded Systems (SIES), 2011. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008. International Workshop on Worst-Case Execution-Time Analysis (WCET): 2007, 2009, 2010.

Publications

Conference Articles

1. Michael Zolda, Sven Bünte, and Raimund Kirner. Context-sensitive measurement-based worst-case execution time estimation. In 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11), Toyama, Japan, August 2011. IEEE.

2. Sven Bünte, Michael Zolda, and Raimund Kirner. Let's get less optimistic in measurement-based timing analysis. In 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11), Los Alamitos, CA, June 2011. IEEE.

3. Sven Bünte, Michael Zolda, Michael Tautschnig, and Raimund Kirner. Improving the confidence in measurement-based timing analysis. In 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11), Newport Beach, CA, USA, March 2011. IEEE.

4. Michael Zolda, Sven Bünte, and Raimund Kirner. Context-sensitivity in IPET for measurement-based timing analysis. In 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10), Amirandes, Heraclion, Crete, Greece, October 2010.

5. Michael Zolda, Sven Bünte, and Raimund Kirner. Towards adaptable control flow segmentation for measurement-based execution time analysis. In 17th International Conference on Real-Time and Network Systems (RTNS'09), Paris, France, October 2009. INRIA HAL.

6. Vaclav Mikolasek and Michael Zolda. Towards distributed robustness in embedded systems. In Proc. 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09), Estoril, Lisbon, Portugal, June 2009.

7. Uwe Egly, Martina Seidl, Hans Tompits, Stefan Woltran, and Michael Zolda. Comparing different prenexing strategies for quantified boolean formulas. In Enrico Giunchiglia and Armando Tacchella, editors, 6th International Conference on Theory and Applications of Satisfiability Testing, volume 2919 of Lecture Notes in Computer Science, pages 214-228,

Workshop Articles

1. Compiler support for measurement-based timing analysis. In Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET'11), Porto, Portugal, July 2011. OCG.

2. Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Reinhard Wilhelm, Armelle Bonenfant, Hugues Cassé, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. WCET tool challenge 2011: Report. In Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'11), Porto, Portugal, July 2011.

3. Michael Zolda and Raimund Kirner. Divide and measure: CFG segmentation for the measurement-based analysis of resource consumption. In Junior Scientist Conference 2008, pages 117-118, Vienna, Austria, November 2008. Technische Universität Wien.

4. Michael Zolda. INFER: Interactive timing profiles based on bayesian networks. In 8th International Workshop on Worst-Case Execution Time Analysis, Prague, Czech Republic, October 2008.

Book Chapters

1. Raimund Kirner, Sven Bünte, and Michael Zolda. Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility, chapter 5: Measurement-Based Timing Analysis for Reconfigurable Embedded Systems. IGI Global, Hershey, PA, USA, November 2010.

Theses

1. Michael Zolda. Comparing different prenexing strategies for quantified boolean formulas. Master's thesis, Vienna University of Technology, Karlsplatz 13, Vienna, Austria, 2005.