Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Masters's Thesis

Precise Measurement-Based Worst-Case Execution Time Estimation

submitted by

Stefan Stattelmann

on September 22, 2009

Supervisor

Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm

Advisor

Dr. Florian Martin

Reviewers

Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm
Prof. Dr. Bernd Finkbeiner

Abstract

During the development of real-time systems, the worst-case execution time (WCET) of every task or program in the system must be known in order to show that all timing requirements for the system are fulfilled. The increasing complexity of modern processor architectures makes achieving this objective more and more challenging. The incorporation of performance enhancing features like caches and speculative execution, as well as the interaction of these components, make execution times highly dependent on the execution history and the overall state of the system. To determine a bound for the execution time of a program, static methods for timing analysis face the challenge to model all possible system states which can occur during the execution of the program. The necessary approximation of potential system states is likely to overestimate the actual WCET considerably. On the other hand, measurement-based timing analysis techniques use a relatively small number of run-time measurements to estimate the worst-case execution time. As it is generally impossible to observe all potential executions of a real-world program, this approach cannot provide any guarantees about the calculated WCET estimate and the results are often imprecise.

This thesis presents a new approach to timing analysis which was designed to overcome the problems of existing methods. By partitioning the analyzed programs into easily traceable segments and by precisely controlling run-time measurements, the new method is able to preserve information about the execution context of measured execution times. After an adequate number of measurements have been taken, this information can be used to precisely estimate the WCET of a program without being overly pessimistic. The method can be seamlessly integrated into frameworks for static program analysis. Thus results from static analyses can be used to make the estimates more precise and perform run-time measurements more efficiently.

Contents

1 Introduction

1.1 Timing Analysis of Embedded Systems

The use of computer technology increasingly pervades everyday life. Almost every electronic product uses one or more built-in microprocessors to fulfil its purpose. Yet, the presence of a computer system within the final product might not always be noticeable for the user.

Many of these *embedded systems* are subject to time constraints and therefore also called *real-time systems*. They can be separated into two groups: *hard* and *soft* real-time systems. The fulfilment of the timing constraints is mandatory for the correctness of hard real-time systems, i.e. if an operation is not performed within a fixed time frame, the whole system fails. For *soft* real-time systems, however, this is only a desired property, as the system is able to tolerate a small number of time constraint violations. If these violations do not occur too often, the system is still able to resume normal operation, although the quality of the service provided may be degraded.

Most real-time systems comprise several tasks sharing a common resource, e.g. the processor. In order to determine whether all tasks are able to meet their timing requirements, their *worst-case execution time* (WCET) must be known. In the following, the terms *task* and *program* are used interchangeably, both meaning executable code that may be part of a larger system.

There are static and dynamic methods for obtaining the worst-case execution time. Static timing analyses try to determine a safe upper bound for all possible executions of a given program. In contrast, dynamic methods use measurements taken during a finite number of actual executions to determine an estimate of the WCET. The latter approach comes with the risk of underestimating the worst-case execution time because it was not encountered during the measurements. To overcome this problem, measurement-based timing analyses tend to add a safety margin to the WCET estimate to increase the probability that the actual worst-case execution time is bounded by the estimate.

On complex processor architectures, both methods do not always produce optimal results. The optimization of modern processors for average-case performance does not only often deteriorate the worst-case performance of programs, it also makes the prediction of execution times much harder. This is due to the fact that means meant to improve the performance, like caches, pipelining and speculative execution, make the time necessary to execute certain parts of a program highly dependent on the state of the overall system.

As static timing analyses work with an abstract model of the processor, it is computationally impossible to model all possible system states during the execution of a program. Hence the state dependence of execution time has to be resolved by conservative approximations. With the increasing complexity of modern processors, the gap between the actual worst-case execution times and the bounds reported by static timing analyses is therefore likely to grow.

Measurement-based methods suffer from similar problems. The execution time of a few complete runs of a program might highly deviate from the actual worst-case run because the system was always started in a “good” state by chance. Likewise, measuring parts of the analyzed programs with an artificially induced worst-case hardware state (however this may be defined) and combining the partial measurements is unlikely to provide a precise WCET estimate. Modifying the state of the observed system might lead to wrong results because the measurements are taken during hardware states that either never occur in practice or do not lead to a global worst-case execution. Thus, the results of such methods are likely to exhibit a level of imprecision that renders them useless.

For applications that require hard real-time systems, like those in the aerospace industry, measurement-based methods for the determination of worst-case execution times are out of question because they cannot provide any guarantees about the safety of the result. In areas where soft real-time systems are used, for example in multimedia devices or parts of the automotive domain, measurement-based techniques might very well be applicable. Nevertheless, there is a tendency to use static methods in those areas as well, as they provide better results than existing measurement-based methods (as shown for example in [Seh05]).

The inherent cost pressure of these mass-produced items motivates the development of methods which reduce the overestimation of current timing analysis methods. Recently developed methods for precise on-chip execution time measurements encourage a reconsideration of measurement-based analyses for this purpose. This may allow a better utilization of system resources for applications that do not require safe timing bounds and hence reduce the overall system cost.

1.2 Contribution

The goal of this thesis is to investigate whether a context-sensitive analysis of accurate instruction-level measurements can be used to provide precise worst-case execution time estimates. The notion of context-sensitivity is a well-known concept from static program analysis. It has been shown that the precision of an analysis can be considerably improved if the execution environment is considered. This especially holds if the analysis does not only consider different call sites but also distinct iterations of loops (see [Mar99]). Up to now, context information is mainly used in static timing analysis. In this work it will be examined whether measurement-based timing analyses can benefit from the use of context information as well.

Current embedded processors already provide the tools for accurate cycle-level tracing (like Infineon's Multi-Core Debug Solution [MH08]), but no methods that can make use of the high granularity of information provided by those measurements are commonly known. In this work it is to be investigated whether static analyses which determine feasible program paths and classify the cache behavior of programs, combined with context-sensitive measurements, will be able to provide precise estimates of the worst-case execution time. Since the process of tracing can very easily be automated, a large number of context-sensitive measurements can effortlessly be gathered. The incorporation of context information should enable this new method to capture the timing behavior of complex architectures without being overly pessimistic.

It is believed that one of the keys to more precise measurement results is the distinction of loop iterations. When a loop is executed for the first time, the performance is likely to suffer because the body of the loop is not yet in the instruction cache. For the remaining iterations, the loop probably *is* in the instruction cache and hence it will be executed faster. A timing analysis that is not able to make this differentiation will most likely overestimate the overall execution time. Furthermore, the number of times a loop is executed in a routine often depends on the parameters of the routine. The arguments in turn might deviate at different call sites of the routine. Hence the inability of any timing analysis to consider different call sites of a routine is another source of imprecision.

Thus, investigating the adoption of methods from static timing analysis to measurement-based approaches seems quite promising with the increasing availability of tools for precise instruction-level measurements. As recent developments in debug hardware technology offer the possibility to control cycle-accurate traces with a fully programmable on-chip event logic, which can be used to encode state machines for example, precise timing information can be gathered from runtime measurements. An accurate method for measurement-based WCET estimation could be adapted to new processor architectures much more easily than the models used in static timing analysis. Hence, successful research in this area might be able to reduce the initial investment necessary to apply exact timing analysis methods to new architectures.

1.3 Overview

The remainder of this work is organized as follows: chapter 2 presents existing methods for the analysis of worst-case execution times. The theoretical foundations and some notation used in the following chapters are introduced in chapter 3. A formal definition of the method presented in this work can be found in chapter 4. The actual implementation of the method is outlined in the next chapter. In chapter 6 the application of the method to test cases on the Infineon TriCore TC1797 microprocessor is described and evaluated. Chapter 7 gives a summary of the thesis and describes possible extensions.

2 Existing Methods

2.1 Approaches for Timing Analysis

Timing analysis tries to determine limits for the execution time of a task. Results can be lower limits, meaning the *best-case execution time* (BCET) of the task, or upper limits, meaning the *worst-case execution time* (WCET) of the task. The times reported by an analysis can be either safe *bounds* that are guaranteed to hold, or *estimates* that might not always hold, e.g. because the analysis is not able to consider every possible execution of the program. The latter is typical for *measurement-based timing analysis*, whereas safe bounds can in general only be determined by *static timing analysis*.

The analysis of timing properties is often decomposed into smaller steps in order to make the problem less computationally challenging. Analyses often consist of a *low-level analysis* that determines the timing of smaller units of the analyzed task. In the subsequent *high-level analysis* the timing information of smaller units is combined to the WCET of the overall program. Similar techniques can be applied in the latter phase for static as well as for measurement-based methods since they rely on the structure of the program, e.g. by explicitly or implicitly searching for the worst-case path through the program. The low-level analysis on the other hand, which can also be decomposed further, is highly dependent on the approach which was chosen to determine execution times.

In the following section some existing methods for software timing analysis will be described. This chapter only presents a summary of methods which are related to the analysis that will be presented in the following chapters. A much broader overview of existing approaches for timing analysis can be found in [WEE$^+$08].

2.2 Static Timing Analysis

Static timing analysis methods derive bounds for the execution time of a program without actually executing code on real hardware. Since the execution time of an instruction is no longer a constant on current processor architectures, an abstract model of the processor is required to determine the maximal execution time of an instruction sequence. This model has to reflect the internal state of the processor (e.g. the content of the cache) to determine an upper bound for the execution time.

There are approaches, like the one described in [Wil05], which separate the timing analysis into successive steps. The first analysis step determines possible register value. Then the pipeline and cache behavior is determined, which results in timing information for the basic blocks[1] of the program. In the last phase the longest program path (in terms of execution time) is identified, which yields the WCET.

An alternative to this approach is an integrated analysis that conducts the microarchitectural simulation and the path analysis at the same time (see [Lun02]). But in terms of time complexity, an integrated solution seems to be worse than a partitioned approach.

All methods that require a model of the analyzed processor suffer from the problem that processor manufacturers do not publish complete specifications of their products. Although this model only has to consider elements of the architecture that are relevant for the execution time of programs, the lack of specification makes the development of such abstract models tedious and error-prone. That is because there are no methods to automatically generate or verify microarchitectural models, though there is some ongoing research (e.g. [PSM09, SP07]). Hence, large parts of those models must be written by hand, following the documentation of the manufacturer, which is likely to be incomplete. The model then has to be tested by comparing its results with execution times measured on real hardware. As a consequence of this complex process, the development of processor models is expensive and time-consuming. The process must be repeated for each new processor architecture, and the correctness of the model cannot be guaranteed because there is no formal way to verify it.

2.3 Measurement-Based Timing Analysis

Measurement-based methods rely on gathering timing information while executing the analyzed program. As the number of possible paths through a program becomes very large for complex examples, exhaustive measurements are not possible. So a small number of end-to-end measurements of a task is unlikely to capture the worst-case behavior. Nevertheless, the observation of complete program runs only is still industry practice.

To overcome the problem of exhaustive measurements, several solutions have been developed (e.g. [DP07] and [WKRP08]) that try to partition a program into parts which can be measured more easily. These approaches usually assume that the system can be brought into a worst-case state before taking measurements, e.g. by clearing the cache. This assumption may hold on simpler processors, but it is hard to fulfil it on recent architectures that can exhibit *timing anomalies*.

Intuitively, a *timing anomaly* occurs when the overall execution time benefits if parts of the program are executed more slowly. In other words, a local worst-case execution does not necessarily lead to the global worst-case. The problem will only be sketched here,

[1]A basic block is a sequence of instructions in which the control flow enters only at the beginning and leaves at the end with no possibility of branching except at the end.

a precise classification of timing anomalies can be found in [RWT^{+}06]. In general the reason for timing anomalies are an unwanted interference of (or between) performance enhancing features of the processor. For example, a cache hit can improve performance because it may allow the speculative execution to work correctly. Whereas a cache hit in the same situation would degrade performance because it would result in a branch misprediction whose effects have to be undone.

Another proposal to increase the parts of a program that are covered by measurements is to generate inputs so that all paths are taken [WRKP05, WKRP08]. Although this might be feasible if the measurements must not necessarily take place "in the field", it is probably quite hard, e.g. for systems that rely on complex sensor data, to generate a *realistic* behavior of the analyzed task.

An alternative approach proposed in [BCP02] and [BCP03] is to determine the probability distribution of execution times for parts of the program instead of trying to enforce and measure a worst-case execution. What can be derived from this distributions is not a bound for the execution time, but an estimate for which there is a high probability that the task complies with it. Although the approach seems to consider the interdependency of partial measurements to some extent, it is not clear if it is able to treat architectures with timing anomalies correctly.

All current measurement-based methods for timing analysis suffer from the problem that it is not easy to extract detailed timing information during the execution of a program *on the real hardware*. Although recent developments in this area provide the means to precisely control the generation of cycle-accurate measurement data non-intrusively, this technology is not available for all current embedded processors yet. Hence it might by necessary to add instrumentation code to an executable before measurements can be taken on the target hardware for some platforms. Other approaches rely on timestamps generated by running the task on a cycle-accurate simulator instead of the real processor. The latter approach has downsides that are similar to those of microarchitectural models used in static timing analysis: it might be hard to verify that the behavior of the actual hardware and the model coincide if the later is not synthesized from a formal specification.

The addition of instrumentation code raises the question how this modification influences the timing behavior of the analyzed task. Adding the instrumentation code during measurements may result in a different timing behavior of the analyzed program and the production code. On the other hand, leaving the (deactivated) instrumentation code in the final executable might worsen memory and power efficiency. Therefore many of the proposed methods for measurement-based analysis try to minimize the number of necessary instrumentation points without losing too much precision. One recent example is [BB06], but similar research has already been done in the area of program profiling [BL92].

2.4 Hybrid Methods

Several researchers claim that their methods for worst-case execution time determination are *hybrid methods* because they use a combination of static analyses and measurements. In the present work, this classification will not be used. Instead, the only distinction that will be made is between static timing analysis and measurement-based timing analysis. The so called hybrid methods still get the information about the execution times by measuring program runs. Static methods are merely used to analyze the structure of the program to be measured. But this is obviously necessary for all measurement-based methods that want to do more than end-to-end or function level measurement. Hence calling such methods *hybrid approaches* is misleading in most cases because no static timing properties are determined, but the timing behavior is still only measured.

An exception is the combination of a simple cache analysis and measurements as proposed in [SSPH06]. The static cache analysis determines the possible number of cache misses per basic block and hence determines potential bottlenecks of the program execution. This information is then used to verify that the taken measurements suffice to describe the task's timing behavior by checking that there is a correlation between the predicted number of cache misses and slower execution times. If this is not the case, the paper proposes a feedback mechanism to measure those parts of the program again whose measured behavior deviates from the prediction. This is meant to increase the probability that the worst-case behavior is covered by the measurements used to estimate the WCET.

2.5 Context-Sensitive Tracing

To improve the precision of measurement-based timing analysis, the *execution context* of the analyzed code must be taken into account. The first attempt to consider the execution context was the *structure-based approach* originally proposed by Shaw in [Sha89]. But it only aimed at a more precise combination of execution times from program parts that were measured separately. Similar methods are still used, e.g. by [BCP03], but they lack the ability to reflect the interaction of individual program parts.

Context-sensitive analyses are a well-known technique in *data flow analysis* to improve the precision of the results. This is done by taking the *calling context* into account when analyzing the body of a routine, e.g. by incorporating an abstract representation of the call stack that would occur during an actual execution. For measurement-based methods, the term is not as well-defined.

Uses of the term *context-sensitivity* in measurement-based timing analysis include considering flow-constraints, as in the structure-based approach, input-dependent path exclusion like in [WE01], but also variable execution times for different call sites of functions [BB00] and loop iterations [BB06], or a combination of flow and time constraints [MB08]. Therefore one must not confuse the different definitions of a *context-sensitive analysis* when comparing various methods.

In the remainder of this work, the notions of *contexts* from data flow analysis as described in [Mar99] will be adopted. The context-sensitive information gathered by the measurement-based timing analysis described in the following chapters will not only consider different call sites of an analyzed routine, but also different iterations of loops. This approach was able to improve the precision of static analyses considerably and will hopefully do the same for a measurement-based approach. The adoption of techniques from data flow analysis to measurement-based methods has already been done with very good results in [ABL97], though for a slightly different purpose.

3 Foundations and Definitions

3.1 Control Flow Representation

This section defines the representation of program control flow used in the remainder of this work. All analyzed programs are assumed to be in compiled binary format. Hence the control flow is looked at on the level of machine instructions. The definition used here is based on the one given in [The03], but several details that are not used in this work were left out. In particular, this section will only introduce the program representation, but not how to extract it from the executable program. Extensive work in this area has been done by Theiling in [The03].

To illustrate some of the concepts, the example program in figure 1 will be used. It is written in pseudo assembler. The calling convention for the example are that routines receive their arguments in the register `r30` and return the result in the same register. Calling routines and returning from the callee to the caller is handled by the instructions `CALL` and `RET`. The `BEQZ` instruction is used to implement conditional jumps. The meaning of the other instructions in the program should be obvious, but the precise semantics do not matter for the purpose of the example.

```
0x4000 test:  MOV r3 r30
0x4004        CALL addnb
0x4008        BEQZ 0x4014 r30
0x400C        MOV r4 r30
0x4010        CALL addnb
0x4014        RET
0x4018 addnb: LOAD r30 r1
0x401C        INC r30
0x4020        LOAD r30 r2
0x4024        ADD r1 r2 r30
0x4028        RET
```

Figure 1: Example program

3.1.1 Basic Block

A basic block is a sequence of instructions in which control flow enters only at the beginning and leaves at the end with no possibility of branching except at the end. Let $\mathbf{V}$ designate the *finite* set of basic blocks for the whole program. Each basic block has a unique start address from the set of instruction addresses $\mathbf{A} \subset \mathbb{N}$ and there is a function

$$\mathit{address} : \mathbf{V} \to \mathbf{A}$$

to calculate the start address. Instructions will only be considered in terms of their address. To calculate the respective basic block of an instruction address, the function

$$\mathit{address}^{-1} : \mathbf{A} \to \mathbf{V}$$

is used. Note that a basic block may contain more than one instruction and that is why $\mathit{address}(\mathit{address}^{-1}(a)) = a$ does *not* hold for all addresses a.

3.1.2 Routines

Most programs are structured into smaller, reusable pieces called *functions*, *procedures* or *routines*. To avoid confusion with mathematical functions, the term routine will be adopted here. Let $\mathbf{R}$ denote the finite set of routines for a given program. Every routine $r \in \mathbf{R}$ consists of a fixed number of basic blocks from the set $\mathbf{V}_r$, whereas there is one unique routine for each basic block. In other words, for $r_1, r_2 \in \mathbf{R}$:

$$r_1 \neq r_2 \Longrightarrow \mathbf{V}_{r_1} \cap \mathbf{V}_{r_2} = \emptyset$$

Hence there is a function

$$\mathit{routine} : \mathbf{V} \to \mathbf{R}$$

that assigns the parent routine to each basic block. Furthermore it will be assumed that there is one unique entry block for every routine so that this basic block is the first one to be executed for every invocation of the routine. The set of entry blocks is denoted by $\mathbf{Starts} \subseteq \mathbf{V}$ and there exists a function

$$\mathit{start} : \mathbf{R} \to \mathbf{Starts}$$

that maps each routine to its entry block. Analogously, there is a set $\mathbf{Ends} \subseteq \mathbf{V}$ and a function

$$\mathit{end} : \mathbf{R} \to \mathbf{Ends}$$

for the unique block of a routine through which the routine is left. Since there might be more than one instruction per routine that results in leaving the routine, these blocks might be added artificially.

3.1.3 Control Flow Graph

A *control flow graph* (CFG) is a representation of the possible control flow through a program in terms of a directed graph. The first step is to construct the CFG for every routine in the program of interest. For some routine r, the control flow graph

$$CFG_r := (\mathbf{V}_r, \mathbf{E}_r)$$

consists of the basic blocks of r and the set of edges $\mathbf{E}_r \subseteq \mathbf{V}_r \times \mathbf{V}_r$ that represent the order in which the basic block can be executed during an execution of the program. Control flow graphs that only represent control flow on the level of routines are also referred to as *intraprocedural* control flow graphs. An example for such a graph can be found in figure 2.

3.1.4 Loops and Recursion

One distinctive property of the control flow representation is that loops are treated like recursive routines. Loops are assumed to be extracted from their parent routine during the construction of the control flow graph. During this *loop transformation* phase a special routine is created for each loop in the analyzed program. Though this requires an additional step during control flow reconstruction, uniform handling of loops and recursion is more general and allows the analysis of loop iterations to be separated from the analysis of the parent routine.

To keep the number of paths through the program finite, the maximal number of iterations for every loop and the maximal recursion depth for every recursive routine must be known. This information must be provided either by a user annotation or as the result of a program analysis. For non-recursive routines, the iteration bound is always known to be 1. From this information the function

$$iterations : \mathbf{R} \rightarrow \mathbb{N}$$

can be constructed which will be used in the remainder of this work to determine the iteration bound for a given routine.

3.1.5 Contexts

In order to consider the execution context during the analysis of a program, the control flow graph is extended with additional information that represents the execution context of a routine. One way to do this is to use a *call string* to model the routine's execution history. Call strings can be seen as an abstraction of the call stack that would occur during an execution of the program. In this work, a call string will be represented as a sequence of elements from the set $\mathbf{C} \subseteq \mathbf{V} \times \mathbf{R}$ of call string elements. A call string element $(b, r) \in \mathbf{C}$ describes a call from the basic block b to the routine r. Only *valid* call string elements will be allowed, meaning it must be possible that the last instruction of the

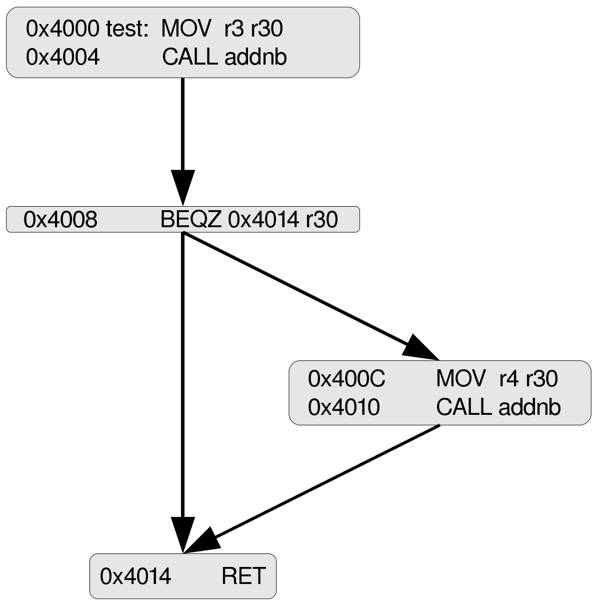

Figure 2: Control flow graph for routine `test` from program in figure 1

basic block b is executed immediately before the first instruction of routine r. Intuitively, the last instruction of b must be a call instruction which can call r. A call string $c \in \mathbf{C}^*$ denotes a sequence of routine calls and the respective call sites. Valid call strings consist only of valid elements and must describe an uninterrupted sequence of routine calls that may call each other. In other words, for valid call strings the following holds

$$c \in \mathbf{C}^* \textit{ is valid} \Longrightarrow \forall \textit{substrings } (b,r)(b',r') \textit{ of } c: \; r = \textit{routine}(b')$$

For the entry routine of the analyzed task (e.g. *main* in a standard C program) there is no execution history as the execution is started by calling the respective routine. This context is described by the empty call string ϵ. The intuition behind this representation of an execution context is that whenever a routine is called, the call string is extended with another element to describe the context of the function body. Therefore extending the call string $c \in \mathbf{C}^*$ with elements from $\mathbf{C}$ works similar to extending the call stack during program execution. Since the execution history of a routine can be very complex, its representation from the set of call strings $\mathbf{C}^*$ might become very long. In order to achieve a more compact representation of execution contexts, the maximal length of call strings will be bounded by a constant $k \in \mathbb{N}_0$ and only call strings from the finite set $\mathbf{C}_k := \{c \mid c \in \mathbf{C}^*, \; |c| \leq k\}$ will be used. For call strings which describe a valid execution but exceed the maximal length, only the last k call string elements will be used to describe the context.

By limiting the call string length, it is not possible anymore to distinguish all execution context that might occur during a program run. Nevertheless, this limitation makes the number of (considered) execution contexts per routine finite, they can be numbered, which makes the representation more general and easy to work with. Information about the potential call sites of a routine and hence its possible execution contexts are determined while reconstructing the program control flow. In the following it is assumed that there are functions to determine information about the valid execution contexts of every routine which are provided by the analysis framework. Let $\mathbf{P} \subset \mathbb{N}$ denote a finite, dense interval from the set of natural numbers. Elements from this set will be used to describe an execution context by an execution context number called *position*. For this purpose, the following functions are assumed to be available:

$$pos : \mathbf{C}_k \times \mathbf{R} \rightarrow \mathbf{P}$$

$$con : \mathbf{P} \times \mathbf{R} \rightarrow \mathbf{C}_k$$

$$contexts : \mathbf{R} \rightarrow 2^{\mathbf{P}}$$

The function *pos* is used to translate valid call strings to valid positions. Assuming a fixed routine r is used, *con* is the inverse function of *pos*. The *contexts* function assigns the set of all valid positions to each routine. The relation of *con* and *pos* is defined as follows:

$$\forall r \in \mathbf{R}\ \forall p \in contexts(r)\ \forall c \in \mathbf{C}_k,\ \ c\ valid\ for\ r :$$

$$con(pos(c, r), r) = c\ \wedge\ pos(con(p, r), r) = p$$

In the remainder of this work, the term *context* will refer to the execution history in which certain parts of a program can be executed. Depending on which representation seems more appropriate, some parts will use the call string representation, others will use execution context numbers (positions). In both cases, the finite representation of a potentially infinite number of possible execution histories will also be referred to as context.

3.1.6 Interprocedural Control Flow Graph

The *interprocedural control flow graph* (ICFG) of a program combines the information from all intraprocedural control flow graphs of a program with the caller/callee-relation between routines. Additionally, the ICFG will be extended with context information. The interprocedural control flow graph of a program is a directed graph with

$$ICFG := (\hat{\mathbf{V}}, \hat{\mathbf{E}}, s)$$

and will be constructed from the set of all basic blocks with context information

$$\hat{\mathbf{V}} := \bigcup_{r \in \mathbf{R}} \mathbf{V}_r \times contexts(r)$$

Furthermore, there is the set of edges $\hat{\mathbf{E}} \subseteq \hat{\mathbf{V}} \times \hat{\mathbf{V}}$ and a start block $s \in \hat{\mathbf{V}}$ from which every run of the respective program is assumed to start. It follows directly from the definition that for every routine and every valid execution context of the routine, the

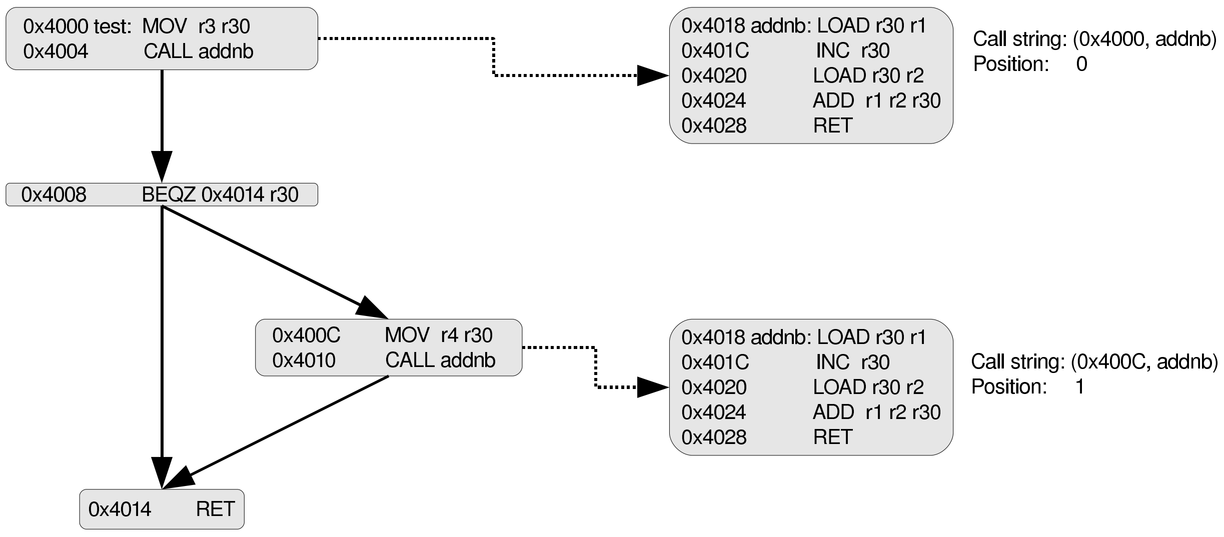

Figure 3: Interprocedural control flow graph for program in figure 1. Solid lines depict the intraprocedural control flow, dotted lines represent call edges.

nodes of the intraprocedural control flow graph with the respective context are part of the ICFG. Furthermore it must hold that

$$\forall r \in \mathbf{R}\ \forall (v_1, v_2) \in \mathbf{E}_r\ \forall p \in \mathit{contexts}(r) : ((v_1, p), (v_2, p)) \in \hat{\mathbf{E}}$$

to ensure no intraprocedural control flow information is lost.

To add information about the caller/callee-relation between routines to the ICFG, let **Calls** $\subset$ **V** denote the set of all basic blocks after whose execution another routine will be entered. Since loops are translated to routines, *call blocks* do not necessarily have to contain any special instructions. Every basic block in front of a loop can become a call block. A *call edge* from the set $(\mathbf{Calls} \times \mathbf{P}) \times (\mathbf{Starts} \times \mathbf{P})$ is used to describe the caller/callee-relation in the ICFG. A very simple example for an interprocedural control flow graph is displayed in figure 3.

Determining a *safe* approximation of the interprocedural control flow of a program is a nontrivial task and may require a number of sophisticated analyses which are out of the scope of this work. Hence, there will be no formal description on how to construct an ICFG with context information for a given program here. Extensive work about how this can be achieved and why the flexible consideration of the execution context is necessary has been done in [The03] and [Mar99]. Instead, the given definitions will be assumed to be available as a toolkit to direct the measurement of program executions and to annotate the timing results to the graph in a context-sensitive way.

The beauty of the definitions presented so far is that there are no restrictions on how many execution contexts can be distinguished for different parts of the program. In particular it is possible to use many execution contexts for some parts of the program, but only a few for others. This is especially useful for loops, since those are treated like recursive routines. Hence different iterations of a loop can be regarded as different execution contexts, not only different call sites of the same routine. It was shown that this approach called *virtual inlining, virtual unrolling* (VIVU) can improve the precision of analyses that employ control flow information of a program considerably [MAWF98].

Intuitively, VIVU applies *function inlining* and *loop unrolling* when constructing the interprocedural control flow graph. This may lead to several copies of the same routine in the graph since the basic blocks for each routine are duplicated for every context. The graph in figure 3 illustrates this by including the basic blocks for the routine `addnb` twice, once for each call site. On the one hand, the number of considered execution contexts, and hence the complexity of the ICFG, is determined by the maximal call string length. For loops, call strings alone are not optimal and a more precise consideration of their execution context is possible by detaching the unrolling of loops from the restrictions of the call string length. This is the actual improvement provided by VIVU, but not the scope of this work. Therefore it will only be assumed that all execution contexts can be represented by *some* finite call string representation, but the maximal length will not be fixed deliberately.

3.1.7 Paths in Control Flow Graphs

Let s and e denote some arbitrary nodes in an interprocedural control flow graph. Possible paths through the graph will be represented by a sequence $p \in \hat{\mathbf{V}}^k, k \in \mathbb{N}$ of nodes in the graph. The notation $s \to e$ will be used for the set of all possible paths between two nodes in the ICFG according to the program control flow. Note that there is no restriction on how often a node may occur on a path in the ICFG. The definition for paths in intraprocedural control flow graphs is accordingly.

3.2 Program Analysis

3.2.1 Data Flow Analysis and Abstract Interpretation

This section is intended to give a short introduction to *data flow analysis*. A more detailed description of data flow analysis and *program analysis* in general, including a formal description of the underlying theory, can be found in [NNH99].

The goal of a program analysis is to statically determine properties that hold for every possible execution of a given program. Since most of these characteristics are undecidable, approximations must be used as correctness and completeness cannot be achieved together. Hence it is generally impossible to derive exact results. Nevertheless, the approximated results of an analysis must be safe with respect to their later use. This means an analysis may sacrifice completeness, but correctness must always be guaranteed. An approximation is considered to be safe if and only if every possible program state is reflected in the analysis results. If the analysis is not able to find out something about the desired property, which means that everything is possible, this information also has to be present in the analysis result. This will sometimes result in *over-approximation.*

There are three main approaches to program analysis: constraint based analysis, type based analysis and data flow analysis, but only the latter will be considered here. A data flow analysis works on the control flow graph of a program and computes an analysis result for each node in the control flow graph. This works as follows: the information the analysis wants to compute is represented by the *analysis domain.* It highly depends on the property that is analyzed. Furthermore, the analysis has to define a *transfer function* that models the effect the execution of a program part has on the analysis domain. To obtain a result, the analysis does a fixed point iteration on the control flow graph. In the beginning, some initial value is assigned to each node in the graph. Afterwards, the analysis goes over the graph and applies the transfer function to each node, combining the new result with the results from previous iterations. This is done until the result has stabilized for each node. The way the analysis selects nodes from the graph and combines old information with new information has a great impact on the performance of the analysis, but the details have been omitted in this brief description.

The theory of *abstract interpretation* [CC77] formalizes this approach by using *abstract values* or *value descriptions* as analysis domain. The way the abstract values change during the analysis is determined by an *abstract semantics.* The analysis domain usually is a simplified version of the property to be analyzed (e.g. value intervals if one is interested in possible register values) which can be computed more easily during the analysis than the *concrete values.* The abstract semantics, meaning the effects a program statement has on the analysis domain, is based on the *concrete semantics*, that is the effect a program statement would have on a concrete value during an actual execution. Since the approach is semantics-based, correctness proofs for program analyses are easily possible. Moreover, the abstract semantics and hence the transfer function of an analysis can usually be derived in a systematic way from the concrete semantics.

3.2.2 Cache Analysis

Caches have a huge influence on the execution time of a program. The execution times of a memory access which can make use of the cache and one that misses it may differ by up to two orders of magnitude. Hence looking at the cache behavior of a program is likely to provide an indication in which parts of it performance may degrade. The inclusion of the execution context of program parts will provide even greater insight. This is the case because caches rely on the principle of locality, i.e. that the same regions in memory are accessed again and again. In particular loops and recursive routines benefit from caches. During their first iteration, they are likely to produce many cache misses, resulting in a slow execution of the code. For further iterations, the cache most likely contains the correct memory areas and hence the execution is faster than for the first iteration.

There are four important parameters to characterize a cache:

- *Capacity*: the number of bytes it may contain.

- *Line size* or *block size*: the size (in bytes) of the contiguous chunks of data which are transferred from memory and stored in the cache in case of a cache miss.
- *Associativity*: the number of possible locations for a single block in the cache. The quotient $\frac{\#lines}{associativity}$ describes the number of *sets* in the cache.
- *Replacement policy*: determines which cache lines are evicted once the cache is full.

The cache set to which a given memory area is mapped is only determined by its address. If a cache block can reside in any of the cache lines, the cache is called *fully associative.* On the other hand, if there is exactly one possible location for each block, the cache is called *direct mapped.* A *k-way set associative cache* is a cache where every block can be stored in *k* possible locations. Fully associative and direct mapped caches are special cases of k-way set associative caches and so only the latter must be considered. As the different cache sets cannot influence each other, it suffices to look at them independently.

The cache analysis described in [FMW97, Fer97] determines the possible cache states at each program point (each node in the ICFG) by abstract interpretation. This is done by performing two analyses, a *must analysis* and a *may analysis.* The must-cache analysis determines which addresses must be contained in the cache and so accesses to them are guaranteed to produce a cache hit. Accordingly, the may-cache analysis determines which addresses may be stored in the cache. By computing the complement of these addresses, it is possible to derive which addresses are guaranteed to produce a cache miss. The effect of an access to addresses for which neither a cache hit nor a cache miss can be guaranteed is unclassified and thus both cases are possible.

For each cache set of the cache, the analysis must safely approximate the effect of the possible accesses at each program point. To do this, the destination of memory accesses must be determined in order to know which sets are affected by an access. Determining the destination of a memory access is relatively easy for instructions because, except for computed routine calls or branches, the order in which instructions get executed can be extracted from a binary program in a straightforward fashion. Thus the potential accesses to instruction memory can be determined easily, especially since reconstructing the control flow of a program is a prerequisite for data flow analysis. However, predicting data accesses is not that easy in most cases. Depending on the structure of the program, data accesses might involve sophisticated computations or indirect accesses, e.g. because the program was originally written in C and made use of several levels of pointers. Consequently, the effort necessary to precisely analyze the behavior of data caches is considerably higher. Therefore separate instruction and data caches are easier to analyze (i.e. provide better analysis results) than a single cache which is used for both. The replacement policy of the cache must be modeled as well and it has a great impact on the quality of the analysis results (see [RGBW06]). Deterministic strategies like Least Recently Used (LRU) or First In, First Out (FIFO) replacement provide better results because their behavior can be predicted more easily. Other policies like a pseudo-random strategy or a cache design where the replacement logic is shared by several caches makes the behavior much harder to predict.

Further details of the cache analysis implementation will be omitted. In the remainder of this work, it will be assumed that the results of a cache analysis for the instruction cache are available. For this purpose, the following functions will be used to access these results:

$$ah : \hat{\mathbf{V}} \rightarrow 2^{\mathbf{A}}$$
$$am : \hat{\mathbf{V}} \rightarrow 2^{\mathbf{A}}$$
$$nc : \hat{\mathbf{V}} \rightarrow 2^{\mathbf{A}}$$

The function *ah* returns the set of instruction addresses of a given node in the ICFG for which a cache hit can always be guaranteed. Likewise, *am* yields the addresses for which an access will always result in a cache miss. All other addresses, meaning instructions for which the cache behavior cannot be classified, can be accessed with the function *nc*. The sets returned by these three functions for a given node must be pairwise disjoint. Additionally, they must cover the addresses of all instructions in the respective basic block.

$$\begin{gathered} \forall \hat{v} = (v, p) \in \hat{\mathbf{V}} : \\ ah(\hat{v}) \cup am(\hat{v}) \cup nc(\hat{v}) = \{a \mid a \in \mathbf{A},\ address^{-1}(a) = v\} \ \wedge \\ ah(\hat{v}) \cap am(\hat{v}) = \emptyset \ \wedge \ ah(\hat{v}) \cap nc(\hat{v}) = \emptyset \ \wedge \ am(\hat{v}) \cap nc(\hat{v}) = \emptyset \end{gathered}$$

3.3 Finite State Transducers

The formalism for controlling program measurements will be based on *Mealy machines*, originally introduced in [Mea55]. Similar definitions can be found in any introductory textbook on theoretical computer science.

Mealy Machines

A *Mealy machine* $M = (S, s_0, I, O, f, g)$ consists of a set of *states* S, an initial state $s_0 \in S$, an *input alphabet* I, an *output alphabet* O, a *transition function*

$$f : S \times I \rightarrow S$$

and an *output function*

$$g : S \times I \rightarrow O$$

The sets S, I and O must be finite. Mealy machines translate a string of symbols from the input alphabet into a string of output symbols. This translation is done by considering the current state as well as the current input symbol. Let $i_n \in I$, $n \in \mathbb{N}_0$ be the nth element of the input string. Furthermore, let $s_n \in S$ be the state before reading the nth input symbol and let $o_n \in O$ denote the nth output symbol. State transitions and the output of the machine are then defined as follows:

$$s_{n+1} = f(s_n, i_n)$$
$$o_n = g(s_n, i_n)$$

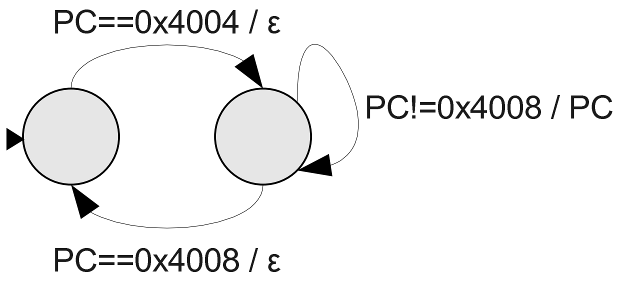

Figure 4: Trace automaton example

Trace Automata

A *trace automaton* $T = (S, s_0, \mathbf{A}, \mathbf{A}, f, g)$ is a Mealy machine that describes which instructions should be traced during a run of a program for which we want to do measurement-based timing analysis. The addresses of the instructions that are executed during a measurement run are the input for a trace automaton. The output symbols are those addresses for which data is stored in the trace buffer. Since the size of the trace buffer is limited, for most of the instructions the address will not be stored. Therefore the symbol ϵ will be used to indicate that no information should be stored in the trace buffer for the current state of the automaton. Furthermore f and g can be partial functions. For the case that the result of an input pair is undefined for one of the functions, it will be assumed that no state transition occurs and no output is produced respectively.

Consider the example program from figure 1 which calls a routine `addnb` from address `0x4004`. The return address for this particular call is `0x4008`, but there are also other calls to this routine that must not be contained in the trace which is to be created. This can be achieved by searching for the address `0x4004` in the sequence of executed instructions of a given program run. Afterwards, all instruction addresses are stored up to the next occurrence of `0x4008`. The trace automaton T depicted in figure 4 implements this task, its output for a valid sequence of instructions is presented in figure 5. The formal definition of the automaton in figure 4 is as follows:

$$T = (\{s_0, s_1\}, s_0, \mathbf{A}, \mathbf{A}, f, g)$$

$$f(s,a) = \begin{cases} s_1 & \textit{if } s = s_0 \wedge a = \texttt{0x4004} \\ s_0 & \textit{if } s = s_1 \wedge a = \texttt{0x4008} \\ s & \textit{otherwise} \end{cases}$$

$$g(s,a) = \begin{cases} a & \textit{if } s = s_1 \wedge a \neq \texttt{0x4008} \\ \epsilon & \textit{otherwise} \end{cases}$$

instruction	address	trace
1	0x4000	
2	0x4004	
3	0x4018	0x4018
4	0x401C	0x401C
5	0x4020	0x4020
6	0x4024	0x4024
7	0x4028	0x4028
8	0x4008	
9	0x400C	
10	0x4010	
11	0x4018	
12	0x401C	
13	0x4020	
14	0x4024	
15	0x4028	
16	0x4014	

Figure 5: Result for the automaton in figure 4 and the program in figure 1

3.4 Notation

In the remainder of this work, the following shortcuts will be used to make formulae and descriptions of algorithms more concise:

Function Updates

Let $f, f' \in X \to Y;\ x, a \in X$ and $b \in Y$. The function update operator $\setminus$ is defined as

$$f' = f\ \setminus \{a \mapsto b\} \qquad \Longleftrightarrow \qquad f'(x) = \begin{cases} b & \textit{if } x = a \\ f(x) & \textit{otherwise} \end{cases}$$

Trace Automata State Transitions

For a more compact representation of trace automata transitions, the function update operator is also defined for functions of type $S \times \mathbf{A} \to S$ and a pair of a state s and a set of addresses $\hat{A} \subseteq \mathbf{A}$.

$$f' = f\ \setminus \{(s, \hat{A}) \mapsto s'\} \qquad \Longleftrightarrow \qquad f'(\hat{s}, a) = \begin{cases} s' & \textit{if } a \in \hat{A} \wedge \hat{s} = s \\ f(\hat{s}, a) & \textit{otherwise} \end{cases}$$

Tuple Decomposition

$$t := (t_1,\ t_2,\ \ldots\ ,\ t_n)$$

$$(v_1,\ v_2,\ \ldots\ ,\ v_n) \leftarrow t \qquad \Longrightarrow \qquad \begin{array}{c} v_1 = t_1 \\ v_2 = t_2 \\ \ldots \\ v_n = t_n \end{array}$$

4 Context-Sensitive Measurements

4.1 Overview

This chapter introduces the theoretical concepts of a new technique for measurement-based timing analysis. The method works on the interprocedural control flow graph (ICFG) of a program executable and requires measurement hardware that can be controlled by complex trigger conditions. The development of the approach was motivated by the limited size of trace buffer memory which is available in current hardware for on-chip execution time measurements. As a consequence of this limitation, it is not possible to determine context-sensitive execution times from end-to-end measurements, since it is not possible to create cycle-accurate end-to-end traces for programs of realistic size. A prototype software to extract execution times from complete program traces has been implemented in preparation of this thesis, but this will not be discussed here. Instead of using traces of complete program runs, this work investigates the use of the programmable trigger logic in state-of-the-art evaluation boards for embedded processors to create context-sensitive program measurements. Current tracing technology, like the Infineon Multi-Core Debug Solution, allows considering the execution history of a program before starting a measurement run. This is achieved by dedicated event logic in the actual hardware which can be used to encode state machines to model the program state. These possibilities motivated the development of an analysis which makes use of this additional logic to generate context-sensitive traces despite the limitations of the trace buffer size.

The analysis is divided into several phases (figure 6).

- Initially, the ICFG is partitioned into *program segments* in such a way that every possible run through the segments can be measured with the available trace buffer memory.
- Additionally, a cache analysis is performed to estimate the number of cache hits and misses for the instruction cache.
- The information gathered during the first two phases is used to generate trace automata that will control the measurements. By using the results of the cache

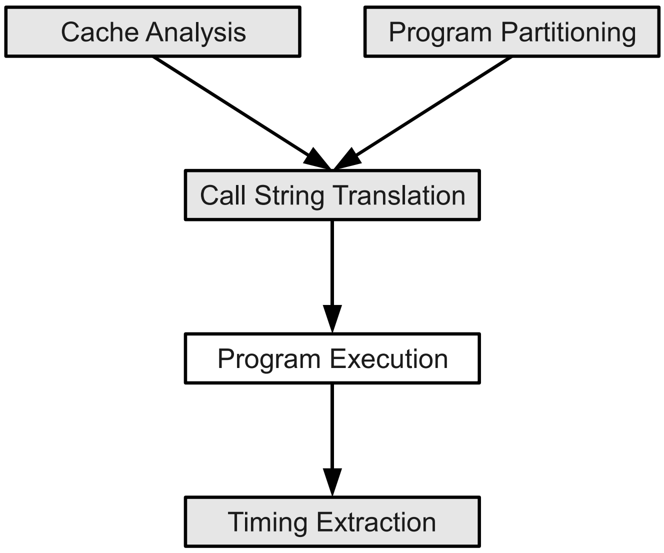

Figure 6: Analysis phases

analysis, it might be possible to join some execution contexts, which reduces the number of measurements that have to be taken.

- Taking measurements requires a sufficiently large number of actual executions of the analyzed program on the target hardware.

- After the measurements have been taken, the context-sensitive timing information for each basic block of the program can be extracted and annotated to the ICFG. Further computations then yield the worst-case path through the ICFG and an estimate of the worst-case execution time of the program.

4.2 Program Partitioning

4.2.1 Motivation

The limited size of trace buffer memory in current measurement solutions prohibits the use of complete end-to-end instruction traces to extract context-sensitive timing information, as for programs of realistic size and even for trivial behavior (e.g. loops with many iterations), the number of instructions executed is unlikely to fit into the trace buffer. Hence, cycle-accurate instruction traces of complete program runs are not possible and therefore the execution context cannot always be reconstructed from the traces.

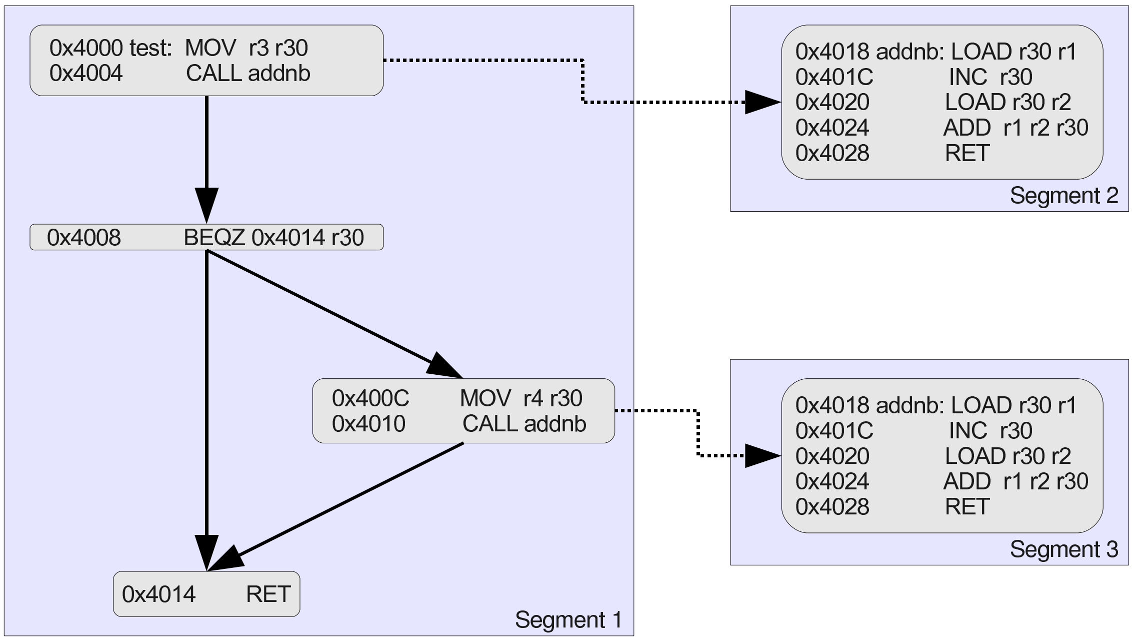

Figure 7: Program segments example

To allow the collection of context-sensitive timing information, even in the presence of a trace buffer with limited size, the program to be analyzed will be partitioned into *program segments*. This partitioning is done on the level of the ICFG and each of the resulting program segments is a part of the ICFG for which every (partial) execution going through it is guaranteed to fit into the trace buffer. Since program segments represent a part of the ICFG, they also include an execution context.

Similar concepts already exist in the literature [WEY01, WRKP05], but the notion of program segments that will be used here is more general and specifically designed to be used for context-sensitive measurements. In particular this means that there will be special segments to measure the body of a routine without calls to other routines which might be contained in it.

Again, the example program from figure 1 will be used to illustrate the concept of program segments. Assume the program is to be traced with a trace buffer which can hold time stamps for at most 6 instructions. In order to extract cycle-accurate and context-sensitive timing information, at least 3 program segments are necessary. Each of these segments is measured individually and the results are combined during a post-processing phase. Figure 7 illustrates one possible partitioning. In this example, a separate segment is created for the body of the routine `addnb` at each call site. Additionally, the segment for the top-level routine `test` is assumed to be measured without the routine it calls. This assumption makes it possible to handle the limited trace buffer. However, to fulfil this assumption during an actual measurement run, it must be possible to control the measurement hardware very precisely.

4.2.2 Definition

A program segment describes instruction sequences on paths in the ICFG. It consists of a start node, an end node and an execution context which has to be the same for both nodes. Additionally, a segment descriptor will be used to classify for which nodes on the paths between the start and the end node trace data should be collected.

Let **SegmentClass** $:= \{GLOBAL, LOCAL\}$ denote the set of program segment descriptors. Segments of type $GLOBAL$ will be used to measure instructions which are executed on the program paths between the start and the end node of a segment. The $LOCAL$ segments are used to only measure the instructions of a single routine, i.e. they are used to trace a routine without the calls it might execute. Program segments are described by tupels from the set

$$\mathbf{Segments} := \{(s, e, p, c) \mid \exists r \in \mathbf{R} : s, e \in \mathbf{V}_r, p \in \mathit{contexts}(r), c \in \mathbf{SegmentClass}\}$$

For $(s, e, p, c) \in$ **Segments**, $(s, p) \in \hat{\mathbf{V}}$ is the start node of the segment in the ICFG and $(e, p) \in \hat{\mathbf{V}}$ is the end node in the ICFG. Therefore program segments only consist of paths in the ICFG that start and end within the same routine and the same execution context.

Due to the limitations of the trace buffer memory on the measurement hardware, it might not be possible to include all instructions on the paths from the start to the end node of a program segment in the generated traces. Therefore the segment descriptors are used to describe which of those instructions must be included in the measurements. Let x be the size of the trace buffer in terms of instruction addresses. For an arbitrary part of a program run that passes through the program segment (s, e, p, c) for which s and e are located in the routine r, let $a_0, \ldots, a_{n-1}$ denote the finite sequence of instruction addresses that are executed on this particular path from s to e. The sequence of traced addresses t depends on the class of the program segment and is defined as follows

$$t := \begin{cases} a_0, \ \ldots\ , a_{k-1} & \mathit{if}\ c = GLOBAL \\ b_0, \ \ldots\ , b_{l-1} & \mathit{if}\ c = LOCAL \end{cases} \qquad k = \mathit{min}\{x, n\} \qquad l = \mathit{min}\{x, |b|\}$$

In the first case, the sequence of traced addresses is the first part of the complete address sequence that still fits into the trace buffer. For the second case, the address sequence with $b_i := \varphi(a_i)$ is used to restrict the traced addresses to the address range of routine r. This is described by the projection function

$$\varphi(a) := \begin{cases} a & \mathit{if}\ \mathit{routine}(\mathit{address}^{-1}(a)) = r \\ \epsilon & \mathit{otherwise} \end{cases}$$

Using the projection excludes calls to other routines than r from tracing and hence reduces the amount of trace memory necessary for measuring r context-sensitive. As context information only changes when routines are entered or left, excluding callee routines seems to be a natural approach. It allows the reduction of the memory requirements without the need for a further analysis of the possible control flow in the called routines.

The intuition behind the definition of program segments is that the ICFG will be partitioned into segments so that each node is covered by at least one of them. To allow every instruction contained in a segment to be measured cycle-accurate, segments are constructed such that the maximal length of a trace for a segment is guaranteed to fit into the trace buffer. Since program segments include context information, this allows the extraction of *context-sensitive* timing information for each basic block from the traces of all segments. To achieve this, the traces have to create an exact time stamp for every instruction that is executed. Additionally, it must be guaranteed that measurements for program segments are always taken in the correct execution context. If those conditions are met, this approach overcomes the limitations of a finite trace buffer and therefore makes tighter estimations of the execution time feasible.

4.2.3 Construction

Partitioning a program into traceable segments requires a search for longest paths in the control flow graph. This is done by assigning a weight (the number of instructions) to each basic block. Basically the partitioning algorithm performs a depth-first search (DFS) within the ICFG, but it also makes use of the hierarchical structure of the graph during this process. To determine the maximal length of the path from the start to the end node of each routine, the algorithm starts with calculating the longest intraprocedural path between those nodes. In order to overcome the interdependence of paths through routines that call each other, a fixed point iteration is then used to determine the longest interprocedural path from start to end for every routine in every execution context. This step is also used to determine context-sensitive recursion and iteration bounds, though context-insensitive bounds must be known in advance. This means that all paths in the ICFG are known to be finite. The details of this first step are omitted here but can be found in appendix A.

After the first part of the computation, which provides a mapping *call* for the maximal interprocedural path length from start to end of every routine and a mapping *iter* which provides recursion bounds, the actual partitioning of the ICFG into program segments is done as shown in algorithm 4.1. The algorithm builds on the graph definitions given in the previous chapter and employs the functions *longestLocalPath* and *longestPath* whose exact definitions are given in the appendix. *longestLocalPath* determines the longest intraprocedural path between two nodes. The function *longestPath* determines the maximal length of an interprocedural path between the given nodes, including the maximal number of recursive iterations for every routine which might get called on the path. However, recursive calls to the routine of the start node are not considered in this computation. As a consequence of this, recursion bounds have to be factored in at some points during the partitioning process.

So to partition the ICFG into program segments, the partitioning algorithm is initialized with the start and end node of the entry routine of the graph as initial arguments for the start and end node of the segment that is to be partitioned. It then proceeds as follows:

Algorithm 4.1 Partition the ICFG into program segments with bounded length.

```
partition ($s$, $e$, $m$, $\hat{\mathbf{E}}$, $l$, $call$, $iter$)
    $s$ : start node
    $e$ : end node
    $\hat{\mathbf{E}}$ : edges of the ICFG
    $l$ : maximal length of a path in a program segment
    $call$ : mapping $\mathbf{R} \times \mathbf{P} \to \mathbb{N}$ which contains the length of a routine call per context
    $iter$ : mapping $\mathbf{R} \times \mathbf{P} \to \mathbb{N}$ for context-sensitive recursion bounds of routines

1:  $(s_v, s_p) \leftarrow s$
2:  $(e_v, e_p) \leftarrow e$
Require: $routine(s_v) = routine(e_v) \ \wedge \ s_p = e_p$

3:  $r \leftarrow routine(s_v)$
4:  $m_1 \leftarrow$ empty marking
5:  $m_2 \leftarrow$ empty marking
6:  $m_3 \leftarrow$ empty marking
7:  $\mathbf{E}_r \leftarrow \{(u,v) \mid \exists p \in contexts(r) : \ ((u,p),(v,p)) \in \hat{\mathbf{E}}$
                        $\wedge \ routine(u) = r \ \wedge \ routine(v) = r\}$
8:  $rec \leftarrow true \Leftrightarrow$ path from $s$ to $e$ contains a call to a recursive routine

9:  if $r$ is recursive $\wedge$ $s_v = start(r) \ \wedge \ e_v = end(r)$ then
10:     $factor \leftarrow iter[(r, s_p)]$          // consider recursion bounds in path computation
11: else
12:     $factor \leftarrow 1$
13: end if

14: if $factor * longestPath(s, e, m_1, \hat{\mathbf{E}}, call, iter) \leq l$ then
15:     add segment $(s_v, e_v, s_p, HEAD)$
16: else if $rec = false \ \wedge \ longestLocalPath(s_v, e_v, m_2, \mathbf{E}_r) \leq l$ then
17:     add segment $(s_v, e_v, s_p, LOCAL)$
18:     for all $(r', p') \in \{(r', p') \mid \exists$ edge to $(start(r'), p')$ on some path from $s$ to $e\}$ do
19:         $partition((start(r'), p'), (end(r'), p'), \hat{\mathbf{E}}, l, call, iter)$    // partition called routines
20:     end for
21: else if $r$ is recursive $\wedge$ $2 * longestPath(s, e, m_3, \hat{\mathbf{E}}, call, iter) \leq l$ then
22:     add segment $(s_v, e_v, s_p, HEAD)$
23: else
24:     $middle \leftarrow findDominator(s, e)$
25:     $partition(s, middle, \hat{\mathbf{E}}, l, call, iter)$
26:     $partition(middle, e, \hat{\mathbf{E}}, l, call, iter)$
27: end if
```

- If all instructions on the longest interprocedural path (including a possible recursion) from the start to the end node can be stored in the trace buffer, a valid global program segment has been found.

- For start and end nodes which do not lie in a recursive routine and for which all instructions on the longest intraprocedural path between them do not exceed the maximal length, a local segment can be created. Since local segments do not cover called routines, the partitioning must be applied to all routines that are called on any of the local paths from the start to the end node.

- Loops and recursive routines are likely to be too big to fit completely into the trace buffer with all their iterations. Therefore the partitioning algorithm contains a special case for nodes in such routines. If at least two iterations of the top level recursion (i.e. on the level of the start and end node) fit completely into the trace buffer, the algorithm still creates a global segment for this area of the program. This approach was designed in order to cope with loops or recursions that execute a lot of iterations. Typically, the timing behavior of the first iterations deviates from the others because the content of the cache or other performance enhancing features have not stabilized yet. After some iterations, this should not be the case anymore, which should result in a similar behavior for the following iterations. The intention of this approach is that observing at least the first and second iteration of a loop is still better than being unable to analyze a program at all. Additionally, it is likely that the interesting differences in execution time can already by observed when only looking at the first iterations. Nevertheless, this case has the potential to produce unsafe timing results, e.g. because it is possible that the worst-case execution of a loop body is only observed after a few iterations. That is why it can also be switched off in the implementation of the algorithm.

- If none of the other conditions is met, the size of the program segment must be reduced. This is done by searching for dominator nodes[2] of the current end block. One of the dominator nodes is selected and partitioning continues independently for the start and the dominator node as well as for the dominator node and the end node. If no dominator is found, the algorithm cannot continue and the partitioning will be aborted. By reducing the size of program segments in this way, it is still guaranteed that the start node of each segment is executed for each run through the parent routine. Hence program segments are relatively independent of the paths through the program which are taken during an actual run of it. Therefore it becomes more likely that a segment can be covered (i.e. is executed) during an arbitrary measurement run.

[2]dominator nodes for a given node lie on every path from the start node to the given one

4.3 Trace Automata

4.3.1 Introduction

Partitioning a program into smaller segments is only the first step for taking precise measurements and its main purpose is to overcome the memory limits of current measurement hardware. What is much more important is the ability to precisely control the traces which are assumed to create cycle-accurate timing information without interfering with the analyzed software. The tools for creating these kind of traces exist [MH08], but existing methods for measurement-based timing analysis lack the capability to make use of them in a precise fashion. One of the reasons for this is that measurement-based methods usually are not able to consider different execution contexts in a way that is as precise as in a completely static approach. By considering the execution context already when taking execution time measurements, it is hopefully possible to overcome some of the limitations measurement-based approaches usually have.

The trace automata introduced in section 3.3 are the formalism to describe when to start a trace for a given program segment. Controlling measurements in terms of a state machine allows the incorporation of conditions about the execution history of the measured program parts in an elegant manner. By using the call string representation of an execution context for creating these automata, the context information can be preserved for the subsequent annotation of measured execution times. Therefore a seamless integration of this method into frameworks for static timing analysis is easily possible.

For the description of an execution context, each element of a call string describes two conditions in terms of executed instructions: the call instruction in the call block must be executed immediately before the first instruction of the called routine. Additionally, the sequence of the elements constrains the order of these conditions, i.e. the order of the calls. In principle, they can be directly translated to a trace automaton which changes its state depending on whether the correct routines are called at the appropriate call sites. But since most call sites call exactly one routine, the automata created by this strategy are not minimal. On the other hand, there might be program segments which have a common call string, but lie in a different instruction address range (e.g. if a routine gets partitioned into two segments). Hence it is not sufficient to consider only the context description when constructing trace automata.

4.3.2 Call String Translation

To generate a trace automaton for measuring a program segment $(s, e, p, c) \in \mathbf{Segments}$, the first step is to create states and transitions which correspond to the constraints described by the call string representation $c = \mathit{con}(p, \mathit{routine}(s)), c \in \mathbf{C}_k$ of the segment's execution context. After that, states must be added to express which instructions on the paths through the segment should be traced. The complete approach is depicted in algorithm 4.2 and it proceeds as follows:

Algorithm 4.2 Create the trace automaton for a given program segment.

```
    createAutomaton ($segment$)
        $segment$ : program segment
 1: $(start, end, position, class) \leftarrow segment$
 2: $c \leftarrow con(position, routine(start))$                // translate position into call string
 3: $S \leftarrow \{s_0\}$
 4: $f \leftarrow$ empty function
 5: $g \leftarrow$ empty function
 6: $(n, i) \leftarrow (0, 0)$
 7: while $i < |c|$ do
 8:     $(block, routine) \leftarrow c_i$                    // extract $i$th call string element
 9:     $b_{call} \leftarrow address(block)$
10:     $b_{return} \leftarrow address(block) + \texttt{WORDSIZE}$       // compute return address
11:     $r_{start} \leftarrow address(start(routine))$
12:     $r_{end} \leftarrow address(end(routine))$
13:     $R_{range} \leftarrow [r_{start} : r_{end}]$
14:     $R_{left} \leftarrow \mathbf{A} \setminus R_{range}$
15:     if $block$ can only call one routine then
16:         $S \leftarrow S \cup \{s_{n+1}\}$
17:         $f \leftarrow f \setminus \{(s_n, b_{call}) \mapsto s_{n+1}\} \setminus \{(s_{n+1}, b_{return}) \mapsto s_n\}$
18:     else if $routine$ is only called from $block$ then
19:         $S \leftarrow S \cup \{s_{n+1}\}$
20:         $f \leftarrow f \setminus \{(s_n, r_{start}) \mapsto s_{n+1}\} \setminus \{(s_{n+1}, R_{left}) \mapsto s_n\}$
21:     else
22:         $S \leftarrow S \cup \{s_{n+1}, s_{n+2}\}$
23:         $f \leftarrow f \setminus \{(s_n, b_{call}) \mapsto s_{n+1}\} \setminus \{(s_{n+1}, b_{return}) \mapsto s_n\}$
                $\setminus \{(s_{n+1}, r_{start}) \mapsto s_{n+2}\} \setminus \{(s_{n+2}, R_{left}) \mapsto s_{n+1}\}$
24:     end if
25:     $n \leftarrow |S| - 1$
26:     $i \leftarrow i + 1$
27: end while
28: $t_{start} \leftarrow address(start)$
29: $t_{end} \leftarrow max\{a \mid address^{-1}(a) = end\}$
30: if $class = HEAD$ then
31:     $S \leftarrow S \cup \{s_{n+1}\}$
32:     $f \leftarrow f \setminus \{(s_n, t_{start}) \mapsto s_{n+1}\} \setminus \{(s_{n+1}, t_{end}) \mapsto s_n\}$
33: else if $class = LOCAL$ then
34:     $S \leftarrow S \cup \{s_{n+1}, s_{n+2}\}$
35:     $local \leftarrow [t_{start} : t_{end}]$
36:     $other \leftarrow \mathbf{A} \setminus local$
37:     $f \leftarrow f \setminus \{(s_n, t_{start}) \mapsto s_{n+1}\} \setminus \{(s_{n+1}, t_{end}) \mapsto s_n\}$
            $\setminus \{(s_{n+1}, other) \mapsto s_{n+2}\} \setminus \{(s_{n+2}, local) \mapsto s_{n+1}\}$
38: end if
39: $g \leftarrow \{(s_{n+1}, a) \mapsto a \mid a \in \mathbf{A}\}$
40: return $(S, s_0, \mathbf{A}, \mathbf{A}, f, g)$
```

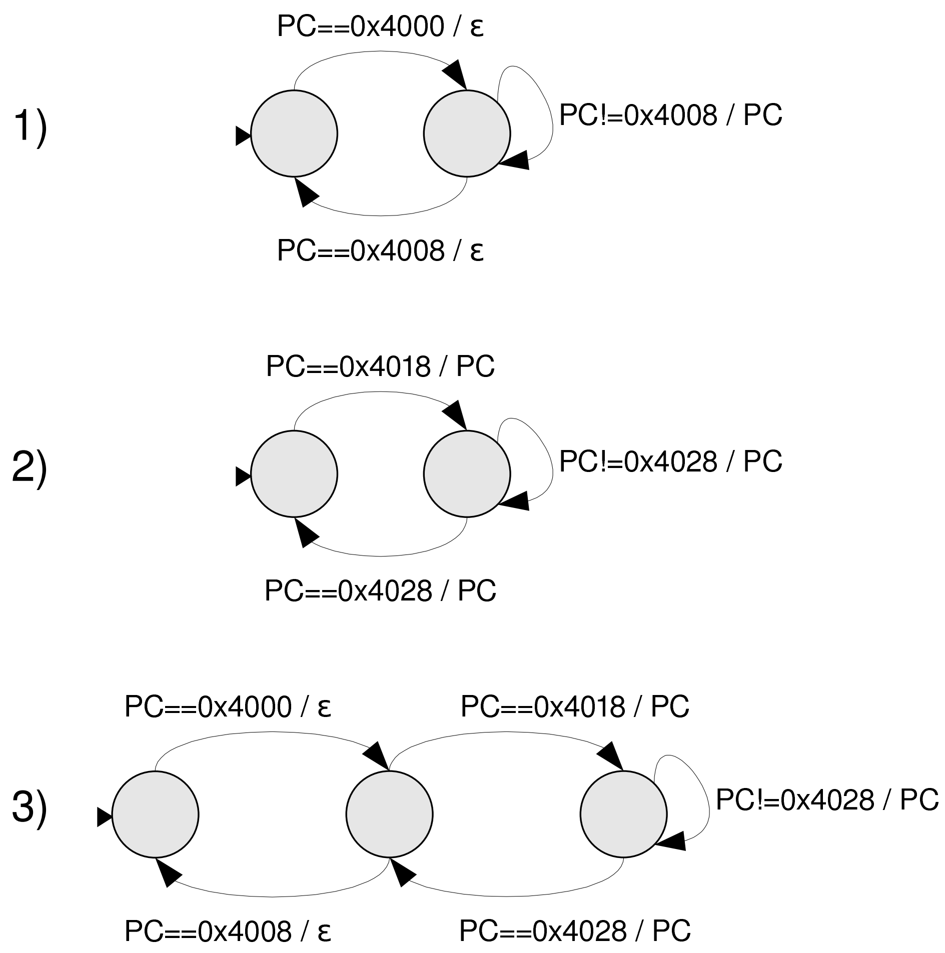

Figure 8: Translation of a call string element to automaton states
1) Basic block calls a unique routine
2) Routine is only called from one basic block
3) Basic block can call several routine and routine has different call sites

- Initially, the automaton has a single state, no transitions and generates no output.

- Then, at least one state for each element of the call string is added to the automaton. How many states are added depends on the properties of the call site described by the string element. If the call described by the element has only one possible destination, it suffices to use the address of the call block as condition for the transition to the next state. Similarly, if this is not the case but the destination routine is only called at this call site, it is enough to add a single state which is entered as soon as the entry address of the respective routine is encountered. For call string elements which fulfil neither of the conditions, both states have to be added to the trace automaton to model the requirements for the execution history described by the call string elements. These three cases are illustrated in figure 8 for the one-element call string (`0x4000`, *addnb*) and the routine `addnb` from figure 3.

- Finally, the state for actually storing trace data is added. For program segments which store global data this can be done by adding a single state which is entered as soon as the start block is entered and left when the end block is left. In this state, all instructions which will be executed will also be stored in the trace buffer. For local segments, things are slightly more complicated and an additional state gets necessary. The tracing state is constructed as before, but the additional state is entered when calls are executed within the segment (i.e. when the address interval for the segment is left) and no trace data will be generated while the automaton is in this state. Note that no extra state for storing trace data is necessary if the program segments covers a complete routine and all its calls. In this case, tracing can start as soon as the addresses for all call string elements of the execution context have been processed by the automaton. This is also illustrated in figure 8.

Algorithm 4.2 only gives an approximation of the steps which are necessary to construct trace automata from program segments. For example, it might not always be possible to compute the return address of a routine call by adding a constant to the address of the call instruction, because the size of instructions is not fixed, but also because the called routine was the result of a loop transformation. The latter case would require additional computations because the return address of a *loop call* is the first address after the loop. Additionally, the way states are created for local program segments might produce automata which do not behave as expected when a segment is not left through its end block, e.g. a routine might have more than one return instruction. This problem can also easily be solved by some additional address computations, but they have been omitted to keep the description more concise.

4.4 Cache Behavior Analysis

4.4.1 Approach

Using the VIVU concept results in a duplication of basic blocks in the ICFG. Due to this duplication of routines for different call sites, the partitioning algorithm might produce several program segments which lie in the same code area. This means there are segments whose start and end nodes have the same address, but a different execution context. Thus the approach presented so far might produce a large number of traces for the parts of the program which are duplicated in the ICFG. As a result of this, some program parts would be covered by a very large number of traces whereas some other parts would not. To achieve a more even trace coverage and to utilize measurement resources more efficiently, the execution context could be discarded for such program segments. Simply using the address range of the respective code regions as prerequisite to start the trace for the segments would reduce the required measurement runs in a very easy way. However, this would also remove the context-sensitivity induced by VIVU and is likely to increase the execution time estimates considerably.

To reduce the number of necessary measurement runs without sacrificing the increased precision offered by context-sensitivity, program segments with a *similar* execution time behavior are joined for tracing. Similarity is looked at on the level of the instruction cache. So two program segments are considered to exhibit similar execution times (in their respective execution context) if the number of cache hits and misses are roughly identical. The cache analysis from [Fer97] which is also described in section 3.2.2 will serve as a foundation for this process. To decide whether two program segments are considered to be similar, *cache behavior metrics* will be used.

4.4.2 Cache Behavior Metrics

A cache behavior metric is a function $m : \mathbf{Segments} \times \mathbf{Segments} \to \mathbb{R}$. Two program segments s and s' are said to be similar with respect to a cache metric m and a similarity constant C if and only if $m(s, s') \leq C$. Cache behavior metrics rely on a previously executed cache analysis as defined in the following.

Since processors in embedded systems often have different types of memory with differing access times, the function *penalty* will be used to classify the severity of a cache hit or miss in a certain address range. For the case that there is only one type of memory, this function can be assumed to always return one. Furthermore, the *weighted* number of accesses for a given node $\hat{v}$ in the ICFG are defined as follows:

$$ah_{\hat{v}} := \sum_{a \in ah(\hat{v})} penalty(a)$$

$$am_{\hat{v}} := \sum_{a \in am(\hat{v})} penalty(a)$$

$$nc_{\hat{v}} := \sum_{a \in nc(\hat{v})} penalty(a)$$

Usually all instruction memory accesses in a basic block should go to the same type of memory. Hence the weighted access numbers just sum up the respective accesses for which the cache is always hit, always missed or for which the effect could not be classified. Eventually this sum is scaled by the penalty factor for the accessed memory range. Based on the access numbers, the distance or *dissimilarity* of two basic blocks $\hat{v}, \hat{w} \in \hat{\mathbf{V}}$ is defined as:

$$dist(\hat{v}, \hat{w}) := \sqrt{(ah_{\hat{v}} - ah_{\hat{w}})^2 + (am_{\hat{v}} - am_{\hat{w}})^2 + (nc_{\hat{v}} + nc_{\hat{w}})^2} \qquad \hat{v} = (v, p) \quad \hat{w} = (v, p')$$

The intuition behind this definition is that the execution time of a basic block in two different execution contexts is roughly identical if the number of cache hits and misses in the contexts is about the same. Note that to conservatively approximate the behavior for unclassified accesses, the weighted number of these accesses is added in the formula. For two program segments $s = (start, end, position, class)$ and $s' = (start, end, position', class)$, their set of differences δ is defined based on the distances between basic blocks in different contexts.

$$s_1 := (start, position) \qquad e_1 := (end, position)$$

$$s_2 := (start, position') \qquad e_2 := (end, position')$$
$$\delta'(s_1, e_1, s_2, e_2) := \{dist(\hat{v}, \hat{w}) \mid \hat{v} = (v, p) \in s_1 \rightarrow e_1,\ \hat{w} = (v, p') \in s_2 \rightarrow e_2\}$$
$$\delta(s, s') := \delta'(s_1, e_1, s_2, e_2)$$

In this work, all cache behavior metrics will be based on these definitions. Nevertheless, the concept of cache behavior metrics could very well be generalized, e.g. to also include knowledge about data accesses. The following metrics were used during the practical evaluation of this work.

Average Distance Metric:

$$m_{avg}(s, s') := \sum_{d \in \delta(s,s')} \frac{d}{|\delta(s, s')|}$$

Maximum Distance Metric:

$$m_{max}(s, s') := max\{d \mid d \in \delta(s, s')\}$$

These metrics were chosen in order to evaluate whether it provides better results if a cache behavior metric considers the whole program segment (i.e. the amortized deviation) or just a single basic block (i.e. the local deviation). Although the metrics were chosen more or less arbitrarily, they seem like a natural choice for this purpose.

4.4.3 Joining of Program Segments

A set of program segments with identical start and end addresses must not be distinguished while taking measurements if they use the same descriptor and the following conditions are met:

- All program segments are similar with respect to a cache behavior metric m and similarity constant C.
- The call strings of relevant program segments share a common suffix of length k. Note that $k = 0$ is a valid choice, but this value can invalidate all context information if the cache behavior metric and the similarity constant are chosen inappropriately.

If these prerequisites are met, it suffices to construct a single trace automaton for the segments using the common call string suffix. The measurements are then taken with the common automaton, but the results are annotated to each segment independently. Thereby the number of trace automata is reduced and the traces generated by a single automaton can be used for several execution contexts of a program part. The second condition is necessary in order to generate a trace automaton which enables tracing in all of the similar execution contexts. Decreasing the value of k increases the probability that this condition is met, but also increases the probability that one of the respective segments does not fulfil the similarity conditions. For this reason, joining of program segments might require user intervention to tweak the values of k and C in order to get optimal results, as good values for these variables depend on the structure of the program to be analyzed.

4.5 Timing Extraction

After a set of traces has been generated for each of the segments into which the program of interest was partitioned, the maximal execution time for each basic block is extracted from the measurements. As the traces are assumed to be cycle-accurate, this is a straightforward process since every instruction which gets executed during a measurement run must also be contained in the respective trace. Additionally, the traces must contain a (relative) timestamp for each instruction. Since tracing is controlled by (an implementation of) a trace automaton, the precise execution context of the trace data is known. Hence the execution time for each basic block can be extracted from a trace by simply going through the trace and the ICFG in parallel. Whenever a new basic block is entered in the trace, the respective node must be found in the ICFG. Depending on the type of program segment which is annotated, this search for a successor must be carried out on the whole ICFG or just within the current routine, i.e. without following call edges. The execution time of a basic block is determined by subtracting the timestamp of its first instruction from the timestamp of the first instruction of its successor block. For each node in the ICFG which was covered by a trace, this provides the execution time for this particular run. Under the assumption that all local worst-cases were observed during the measurements, meaning that the worst-case execution time of each node is covered by at least one of the traces, the maximum from all of the execution times equals the worst-case execution time. All nodes in the ICFG which never occurred in one of the traces are assumed to be never executed during *any* execution of the program. Algorithm 4.3 demonstrates the concept of annotating the execution time from one of the traces to the nodes of the respective program segment.

If a *sufficiently large* number of measurements has been taken under *realistic conditions*, taking the maximum of the measured execution times for each node is likely to provide the worst-case execution time or at least a *realistic estimate* of it. After annotating these execution times to the ICFG, an estimate of the worst-case execution time of the whole program can be computed by *implicit path enumeration.* This method was originally described in [LM95] and it determines the worst-case execution by expressing the potential paths through the program as an optimization problem in terms of integer linear programming (ILP). The resulting constraint system is then solved by an ILP solver. More precisely, the ILP solver determines the maximal sum of basic block execution times for some path from the entry to the exit node of the ICFG. The resulting sum yields an estimate of the program's worst-case execution time. Since this *worst-case path* through the program is determined without considering the program's semantics, it must not necessarily coincide with a path which can occur during an actual execution of the program.

Algorithm 4.3 Extract timing information for a program segment from a trace.

```
    extractTiming ($trace$, $segment$)
       $trace$ : measurement run for the program segment
       $segment$ : program segment
 1: $(start, end, position, class) \leftarrow segment$
 2: $b \leftarrow start$
 3: $a \leftarrow$ first address in $trace$
 4: $t_{new} \leftarrow 0$
 5: $t_{old} \leftarrow 0$

 6: repeat
 7:    while $address^{-1}(a) = b \ \wedge \ trace$ contains data do
 8:       $(a, t_{new}) \leftarrow$ next trace entry
 9:    end while

10:    $executionTime \leftarrow t_{new} - t_{old}$
11:    if annotated execution time for $b < executionTime$ then
12:       annotate $executionTime$ to node $b$
13:    end if

14:    $t_{old} \leftarrow t_{new}$

15:    if $class = LOCAL$ then
16:       $b \leftarrow$ find successor in current routine with address $a$
17:    else
18:       $b \leftarrow$ find successor in ICFG with address $a$
19:    end if

20:    if $b$ is not a valid basic block $\vee$ $address(b) \neq a$ then
21:       abort
22:    end if

23: until $trace$ was read completely
```

5 Implementation

5.1 Overview

The timing analysis method presented in the last chapter was implemented and tested by combining several well-established development tools which are widely used in the automotive and avionic industry:

- The AbsInt aiT Worst-Case Execution Time Analyzers, the leading-edge product for static timing analysis, was used as foundation for extracting control flow and searching for worst-case paths in programs. AbsInt aiT is used e.g. by Airbus to validate the timing behavior of safety-critical parts of flight control software, but also by companies from the automotive domain.

- Infineon Technologies' Multi-Core Debug Solution (MCDS) provided the means for controlling hardware measurements by complex state machines. Furthermore, this technology allows non-intrusive, cycle-accurate measurements of program executions without the need for changes in the analyzed program, like code instrumentation.

- For measuring program executions, the Universal Debug Engine (UDE) from pls Development Tools was utilized. For years, the Universal Debug Engine has been used successfully for developing software components in the automotive industry. Besides the support for MCDS-compatible devices, UDE also provides a special purpose programming language for controlling program traces.

- Infineon's TC1797 microcontroller was used as target platform since it can provide sophisticated means for on-chip execution time measurements. The TC1797 is a 32-bit high-performance microcontroller which is designed for automotive applications and based on the TriCore CPU. It is available in the special variant TC1797ED, which implements the Multi-Core Debug Solution technology.

In the following section, the characteristics of all the components used for the implementation will be introduced. Afterwards their incorporation and the necessary changes to the existing software architectures will be described.

5.2 Tools and Technologies

5.2.1 AbsInt aiT

AbsInt's WCET analyzer aiT [aiT, FH04, Wil05] is a tool for deriving a safe upper bound for the worst-case execution time of tasks in real-time systems. To determine these bounds, which are valid for all inputs of the analyzed task and which consider every path through the program, several microarchitectural analyses are performed. These analyses use abstract interpretation, a semantics-based technique for static program analysis, and require the final binary executable of a program to be able to reason about potential register values and cache contents.

The WCET analysis is divided into several phases (figure 9):

- Initially, the (interprocedural) control flow graph is constructed by decoding the executable. For this purpose, the user can provide an annotation file which contains information about the control flow, like branch targets or loop bounds. Even register or variable contents can be annotated to allow a more precise analysis by excluding certain program paths. Loops in the program are transformed to recursive routines in the ICFG. Furthermore, virtual inlining and virtual unrolling can be used. All subsequent analyses work on the internal CRL representation of the program's control flow graph.
- The first analysis phase determines ranges of possible register values during the task's execution. This information can be used to determine loop bounds from loop patterns found in the program code. All loop bounds which cannot be recognized by this method must be annotated by the user. Additionally, results from the value analysis can also be used to evaluate jump conditions and identify certain branches of the ICFG that will never be executed.
- Next, an integrated cache and pipeline analysis is performed. The cache analysis relies on the address ranges determined by the value analysis to classify whether memory accesses are cache hits or cache misses. Afterwards, this information is used by the pipeline analysis to calculate the maximal execution time of basic blocks by identifying all pipeline states which can occur during the execution of the task.
- Finally, path analysis determines the worst-case path through the program by integer linear programming. For doing this, the path analyzer constructs a system of linear constraints over integer variables which reflect possible paths through the program. Solving this system yields the WCET.

AbsInt aiT is available for a variety of embedded processor architectures. Since the overall toolchain of aiT is very modular, it is not complicated to adopt it to new target architectures. The only exception is the pipeline analysis, as this phase relies on a complex processor model to determine execution times. Like stated before, such models are developed manually and their correctness can only be checked by testing, but not formally

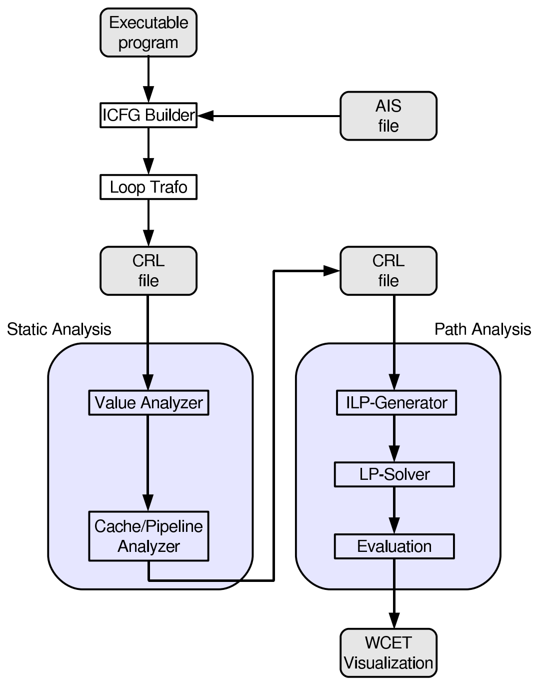

Figure 9: AbsInt aiT toolchain

verified. Hence most of the work for supporting a new target processor in aiT must be done for the development of a new pipeline analysis.

5.2.2 Infineon Technologies Multi-Core Debug Solution

The Multi-Core Debug Solution (MCDS) was developed by Infineon Technologies to address the problem of debugging complex multi-processor systems (e.g. a System-on-a-chip). These systems are often embedded in complicated machinery and faulty behavior may only be observed under real-life conditions. Hence it must be possible to collect trace data from deeply within the system without interfering with its functionality [MH08, IPea].

The approach chosen for the MCDS is to add dedicated logic for collecting trace data on the chip. For each processor or bus that is meant to be monitored in the traces, special control hardware in form of a processor or bus observation block is part of the MCDS. These observation blocks are connected to the units they monitor via special adaption logic to make the internal state of the unit visible to the observation block. Based on this information, the observation blocks can either store trace data directly by sending it to the debug memory controller or generate events for the Multi-Core Crossconnect (MCX) unit of the MCDS. The MCX is equipped with a set of counter registers which can be modified depending on events (or boolean combinations of events) the MCX receives from the observation blocks. It is also possible for the MCX to send signals to the observation

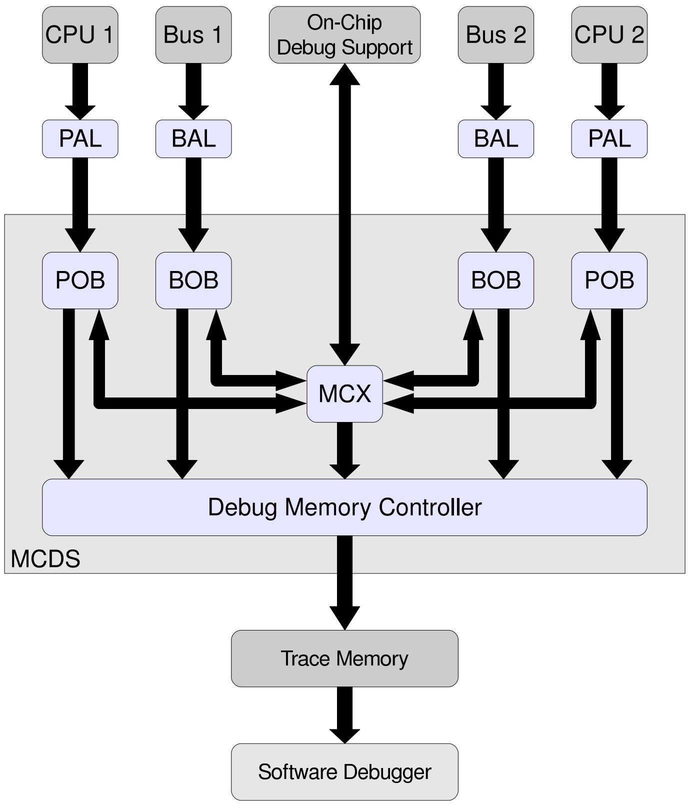

Figure 10: Infineon Multi-Core Debug Solution

blocks depending on the values of the counter registers or other incoming events, but the observation blocks lack the ability to preserve this information because they do not have any storage registers. Hence, more complicated control tasks, e.g. waiting for events happening in a certain order, must be implemented on the MCX. Nevertheless, this design allows to define complex conditions for starting measurements or collecting other data, for example via performance counters. Since the counter registers in the MCX, as well as the events generated by the observation blocks and the MCX, are fully programmable, it is possible to keep track of the overall system by representing it via a state machine. Additionally, the MCX can interact with additional debug support in the system and generate breakpoints for an external software debugger or even halt the whole system. In fact, the whole process of trace data generation relies on a software component which reads the data written to the trace buffer and reconstructs the actual trace from the messages stored by the observation blocks or the MCX.

Figure 10 depicts the interaction of the MCDS and the other components of a system with two processors and two buses[3]. The adaption logic for buses (BAL) and processors (PAL) is not considered to be part of the MCDS. These units only provide information about the internal state of the system to the processor observation block (POB) or the

[3]picture adapted from [IPeb]

bus observation blocks (BOB), but their behavior cannot be influenced. On the other hand, the behavior of the observation blocks and the MCX unit are fully programmable and hence the user of the MCDS has full control over the data that is written to the trace buffer. In the last step of trace generation, the contents of the trace buffer are translated into human or machine readable form by a MCDS-compatible software debugger for further analysis.

Since the additional hardware logic provided by the MCDS is only useful during the development of a product, it is usually left out in the final version. Special variants of the microcontrollers, so called *Emulation Devices* (ED), are used for the development phase, while the standard version without the MCDS will be used in the release versions. These emulation devices are available for a large number of Infineon's microcontroller products and the technology can also be licensed individually.

5.2.3 pls Universal Debug Engine

The Universal Debug Engine [UDE] by pls Development Tools is a software debugger for a large variety of microcontrollers. In particular, the application offers sophisticated facilities for generating program traces. One of the supported interfaces for this purpose is the Multi-Core Debug Solution, i.e. the UDE can serve as the software component of the MCDS. To facilitate the control over the generation of MCDS program traces, the Universal Debug Engine provides the *Trace Qualification Language* (TQL), a special scripting language for easily modifying the on-chip control registers of the MCDS.

By using TQL scripts, the features of the MCDS can be steered in an easy but also very direct way. The generation of events and their exchange between the different components of the MCDS can be controlled very precisely. Figure 11 displays the TQL implementation of the trace automata from figure 4. The conditions for transitions in the automaton are represented by range triggers for the TriCore PC. Thus they are activated when the instruction at the respective address gets executed. The states and transitions are modeled in terms of a counter register in the MCX. This means there is a single counter for both states of the automaton. If the counter is zero, the automaton is in the first state, otherwise it is in the second state. The counter gets modified (i.e. increased or cleared) when the events for the transitions are activated. The second state is also the tracing state, meaning the state where trace data is to be generated. Therefore the value of the counter register must be forwarded to the POB, which is responsible for storing the address of the currently executed instruction (PC). Additionally, the trace must be started at some point. Counter-intuitively, this is done by defining the signal `trace_done` to be activated as soon as the transition to the tracing state is activated. After this event has been signalled, the trace buffer will be filled with the current PC and a time stamp until the buffer is either full or the POB stops storing the PC because the enabling condition (correct automaton state) does no longer hold.

```
// Global configuration
config.memorysize = 0x3ffff;
config.trigger = 0x0;
config.absmode = 0x1;

// Define trigger conditions for the TriCore PC
pob_tc.ptu_trig[0].bound = 0xd4004;
pob_tc.ptu_trig[0].range = 0x2;
pob_tc.ptu_trig[1].bound = 0xd4008;
pob_tc.ptu_trig[1].range = 0x2;

// Triggers that are forwarded to Multi Core Cross-Connect (MCX)
pob_tc.tc_act[0] = pob_tc.ptu_trig[0];
pob_tc.tc_act[1] = pob_tc.ptu_trig[1];

// Automaton states, managed by MCX
mcx.cnt_trig[0].limit = 0x0;

// State transitions
mcx.cnt_trig[0].inc = mcx.tc_act[0];
mcx.cnt_trig[0].clear = mcx.tc_act[1];

// Forward register for tracing state from MCX to POB
mcx.tc_trig[0] = mcx.cnt_trig[0];

// POB stores the PC when in correct state
pob_tc.ptu_enable[0] = pob_tc.tc_trig[0];
pob_tc.ptu_sync[0] = pob_tc.tc_trig[0];

// MCX starts tracing when the tracing state is entered
mcx.trace_done[0] = rise mcx.lmb_act[0];

// Create time stamp for each instruction
mcx.tick_enable[0] = true;
```

Figure 11: Example TQL script

5.2.4 Infineon Technologies TriCore TC1797 Cache Behavior

The TC1797 provides an instruction as well as a data cache. For simplicity, the data cache is completely ignored by the implementation of the cache analysis described in the last chapter, but of course it is enabled during the measurements. The reason for not considering the data cache is that data accesses (and hence possible contents of the data cache) are not as easy to predict precisely as potentially executed instruction sequences. Reasoning about data accesses *precisely* usually requires looking at the behavior of the processor pipeline and this is exactly what the approach presented in this work is meant to avoid. Hence, the data cache is ignored when comparing the cache behavior of program parts in different execution contexts because imprecise information about data accesses seems unlikely to provide any useful information here. In contrast, the content of the instruction cache can be predicted a lot easier and its influence is more obvious, e.g. for different iterations of a loop.

Up to 16 kilobyte of instruction cache are available in the TC1797. The cache is a two-way set associative cache with a Least Recently Used (LRU) replacement strategy and it is organized as 512 cache lines with 256 bits per line. Not all memory segments are cacheable. In addition to the instruction cache, the TC1797 also features a *program line buffer* (PLB). This buffer has the same size as a single cache line. If fetching an instruction located at a cacheable address results in a cache miss, the data to be stored in the respective cache line is read from memory and stored in the PLB. It only gets transfered to the instruction cache after the occurrence of the next cache miss. Program fetch requests for non-cacheable addresses use the PLB as a single line cache. Thereby prefetching is possible for instructions outside of cacheable memory areas.

The implementation of the cache behavior analysis used here only models the instruction cache itself. Despite the nontrivial instruction fetch behavior of the TC1797, it was chosen not to include the PLB in the cache analysis. Similarly, the execution of instructions located in non-cacheable memory segments are assumed to always result in a cache miss. These are valid assumptions because the observation of the cache behavior is *not* done to produce exact numbers for cache hits or misses (or safe approximations thereof). Instead, the cache analysis is done to determine whether the cache behavior of a specific program part varies for different execution contexts. If the analysis is able to detect that this is not the case, it is not necessary to distinguish the execution contexts. Hence always assuming cache misses for certain code areas will classify all contexts as similar. Although always expecting misses might not be an exact representation of what would happen during an actual execution of this program part, e.g. due to the effects of the PLB, this model suffices to classify the behavior correctly. Intuitively this should be obvious since the execution time of instructions from non-cacheable memory areas cannot be influenced by the state of the instruction cache. Similarly, the performance enhancing effect of the prefetching done by the PLB is not relevant for the identification of possible variations of execution times in non-cacheable code areas since this only has a short-term effect and will behave the same for every execution context. Unlike the instruction cache, the PLB is rather independent from the execution history. That is why it suffices to only look at the instruction cache for cacheable code regions for detecting variations in execution time as well.

The drawback of only looking at the instruction cache is that deviations in execution time due to data accesses in different execution contexts cannot be detected. Nevertheless, since for program segments which are considered to exhibit similar execution times, the maximal execution time per basic block seen during a measurement run for all of the execution contexts will be considered to be the worst-case execution time, this will only increase the estimates for the execution time. Since the maximum will be used, this is still a safe approximation, even if tighter bound estimates could be determined with additional measurements.

5.3 Integration of Static and Dynamic Techniques

5.3.1 Incorporation Concept

The new technique for collecting measurement-based timing information which was presented in the last chapter could be integrated into the AbsInt aiT toolchain with reasonable effort. The implementation even extended the concepts and algorithms described in the previous chapter in a few areas to provide even better analysis results. Figure 12 depicts an overview of the changes. In detail, the following modifications and extensions were done:

- The control flow reconstruction phase could be used without any changes as it already comprises the VIVU approach.
- The value analysis was also not altered. This analysis phase might remove parts of the ICFG (i.e. mark parts which are never executed) and is also able to determine loop bounds in addition to the ones already annotated by the user. All further steps are capable of making use of this information.
- In contrast to the integrated cache and pipeline analysis of the original toolchain, an independent cache analysis was added. For its implementation, most parts from the integrated analysis have been reused, but the analysis was restricted to the instruction cache.
- Generating trigger conditions for the measurements incorporates several steps:
  - The ICFG is partitioned into program segments according to the parameters set for the maximal length of the traces. In addition to the partitioning algorithm described in section 4.2.3, several extensions were integrated to allow a better coverage of loop routines.
  - To reduce the number of measurements which have to be taken, it is determined which program segments with same start and end address but different execution contexts are likely to exhibit similar cache behavior with respect to the instruction cache. The decision whether two segments are considered to behave similarly is based on the results of the cache analysis and some additional parameters to control which execution context may be considered to be equivalent. This step only reduces the number of measurements to be taken for the same code area. Intuitively, this step removes some of the duplications in the ICFG which are a result of the VIVU technique.
  - For each group of similar program segments, a trace automaton is created from which a TQL script is generated that controls the measurements in the pls Universal Debug Engine.
  - To make measuring more convenient, a Visual Basic script is created to control the UDE. This script can execute the program to be analyzed and run the TQL

scripts one after another for a given number of repetitions. With this approach, it is possible to start and stop taking measurements without interrupting the program which is traced and there is no need for user intervention. Automating this process allows taking a large number of measurements in a comfortable way. Since the execution of the program is never stopped or restarted, it becomes less likely that different measurements are taken during the same state of the overall system. Thus it is more likely that local worst-cases are covered by the measurements, though a correct estimate of the WCET can only be determined if the program operates under realistic conditions during the measurements. This means that it must still be provided with useful and realistic input data.
  - Furthermore, an XML file containing metainformation is created. This information is used during the annotation of the measured execution times to match the traces to program segments.

- As stated earlier, the pls Universal Debug Engine debugging software is used for the measurements. The tool is controlled by the scripts which are generated during the trigger generation phase. A fixed number of attempts to measure each segment is made. The number of repetitions can be controlled by a parameter, but there are currently no checks whether an attempt to trace a program segment was successful or if a timeout occurred which aborted the measurement. In the current prototype implementation, which uses a TriCore TC1797ED evaluation board, call strings with a maximal length of two are supported. The 256 kilobyte of trace buffer memory can be used to generate cycle-accurate traces of about 100 000 instructions.

- The trace data generated by the measurements is translated into an internal XML format before the timing information is extracted. This step also removes jitter from the traces and slightly reorganizes them for the annotation process.

- In the timing extraction phase, each trace is matched to the respective parts in the ICFG. This provides the analysis with the execution time for each basic block which was covered during the measurements. Additionally, the implementation is also able to derive loop bounds for loops that were completely covered by the measurements. This allows a tighter estimation of execution times under the assumption that the maximal number of iterations occurred during one of the traces. Using this approach to reduce loop bounds is especially useful if very large loop bounds were annotated by the user as a safe approximation because exact loop bounds can neither be found out by a static analysis nor are known to the user.

- After the timing information has been annotated to the nodes in the ICFG, the search for the worst-case path is executed like in the original toolchain. This is done by translating the WCET problem to an optimization problem in terms of an integer linear program. An ILP solver is then used to find the worst-case execution time and the corresponding path in the ICFG. Eventually, it is possible to visualize the ICFG including the worst-case path through the program.

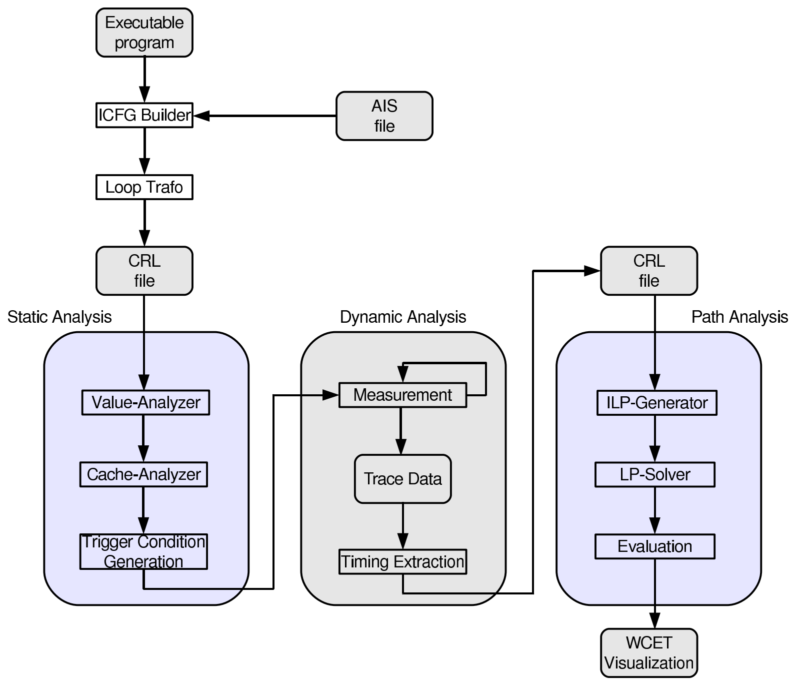

Figure 12: Modified AbsInt aiT toolchain

5.3.2 Problems and Challenges

The limited number of trigger conditions available in current versions of the MCDS imposes some restrictions on the granularity with which execution contexts can be distinguished. Additionally, the combination of events (i.e. preconditions to start a trace) are also limited. As a result of these limitations, only relatively small call strings can be used to describe execution contexts. To overcome some of these limitations, the translation of trace automata to TQL was done in such a way that different parts of an automaton are implemented in different components of the MCDS. This was achieved by moving the logic for starting a trace (the tracing state so to speak) to the processor observation block, while the management of the other states remained on the central MCX unit. Besides, some of the transition computations were moved to the otherwise unused bus observation blocks.

Due to the nature of the MCDS, which in some sense can be seen as a distributed system, there were some delay problems when triggering traces. This has the effect that a trace is missing a few instructions in the beginning. These problems got even worse when the combination of event signals was moved to unused components of the MCDS. Nevertheless, this was necessary in order to allow a more fine-grained representation of execution contexts. In most cases the delay problems could be resolved by adapting the TQL generation, but it could not be solved completely. Furthermore, the delay problems were not always easily reproducible. Especially trigger conditions which are only active for a single cycle seemed to be problematic, e.g. when a trace is meant to be triggered after a single call instruction. Events of this kind were observed to work correctly sometimes, but got lost or were delayed for some of the measurement runs. These problems made the automatic generation of TQL scripts from program segments and call strings quite tedious.

Besides the delay problems when starting a trace, instructions were sometimes left out during traces as well. Though this was not a severe restriction, it required an adaption of the annotation algorithm. Thus the implementation of the execution time annotation does not rely on the assumption that only complete program traces are generated by the measurement equipment. Instead, it suffices if the traces contain a timestamp for at least one instruction of every basic block. As this problem occurred only rarely, the Universal Debug Engine could be used to precisely determine basic block execution times nonetheless.

6 Experiments

6.1 Methodology

6.1.1 Examined Properties

To evaluate the concepts presented in this work, the method was applied to a set of test programs which are based on real-world embedded software. Firstly, this was done to find out how increasing the number of measurements per program segment influences the WCET estimates and what effect the different cache behavior metrics have on them. It was also part of the experiments to check whether the technique is scalable and can be applied to programs of realistic size. Finally, the resulting WCET estimates were compared to the WCET bounds reported by a static timing analysis, to the execution times which were observed during end-to-end measurements and to context-insensitive WCET estimates.

In order to test the robustness and the efficiency of the partitioning and annotation algorithms, the maximal size of the program segments was restricted for most of the test cases. This means that the number of program segments was artificially increased because otherwise many of the test programs could have been traced by a single program segment. By doing this, it was possible to simulate the behavior of the algorithm for very large programs, but the time and space requirements for taking the measurements were minimized. Hence the results presented in the following should also be applicable to programs which are significantly larger than the test cases which were used for the experiments.

6.1.2 Test Setup

The experiments were conducted on an Infineon TriCore TC1797ED evaluation board. For generating program traces version 2.04.09 of the Universal Debug Engine from pls Development Tools was used. WCET bounds for the example programs were determined using the aiT module of the AbsInt Advanced Analyzer a^3 for TriCore, version 9.08i.

To evaluate the effect of the cache on the execution time as well as on the predicted WCET, the code of the test programs was linked to uncached and cached portions of instruction memory. For the first case, programs were stored in the TriCore scratchpad RAM (SPRAM). The same memory is used for the instruction cache, so in effect fetching an instruction from the SPRAM is as fast as fetching an instruction from the cache. Because of the limited size of the SPRAM, this could only be done for small programs. All other examples were stored in a cached memory area of the TriCore program flash. In the following, for all test cases which use the program flash (and hence the instruction cache), this is stated explicitly. If this is not the case, the test program was stored in the SPRAM which results in the same execution time as if every instruction fetch from a cached memory area would produce a cache hit.

6.2 Test Programs

6.2.1 DEBIE-1 Benchmark

The DEBIE-1 (DEBris In orbit Evaluator) satellite instrument is a sensor unit for detecting impacts of small space debris and micro-meteoroids. Its on-board control software was adapted to serve as a benchmark for WCET analysis tools. The software is written in C and was originally developed by Space Systems Finland Ltd (SSF). To serve as a benchmark, the software was adapted by Tidorum Ltd to make it more portable. An internal simulation of I/O devices and an internal test driver were added as well. As a result of these modifications, the benchmark can be compiled for new target architectures with very little effort and the resulting binary already contains code to generate meaningful input data for the original control software. Hence the benchmark is especially useful for measurement-based WCET analysis methods as the system can be observed under realistic conditions without difficulties. SSF provides the DEBIE-1 software for the use as a WCET benchmark under specific terms. A copy of the software can be requested from Tidorum[4].

The control software comprises six tasks. In the original system, these tasks are activated by interrupts and all of them have real-time deadlines. For the benchmark, this behavior is only simulated, i.e. the benchmark is single-threaded. As part of the experiments, the WCET of every task was estimated. This is along the lines of what was done during the WCET Tool Challenge 2008 [HGB+08], for which the DEBIE-1 benchmark was also used. Further information about the purpose of the tasks can be found in the wiki [WCC] of the WCET Tool Challenge 2008.

6.2.2 Mälardalen WCET Benchmark Suite

Three programs from Mälardalen WCET Benchmark Suite [WCE] were chosen for the experiments:

- Edn: The program performs Finite Impulse Response (FIR) filter calculations. This

[4]niklas.holsti@tidorum.fi

involves a lot of vector multiplications, array handling and nested loops. In total, the program contains 12 loops.

- Nsichneu: Program for simulating an extended Petri Net. The source code was automatically generated and contains more than 250 if-statements.
- Statemate: Automatically generated code which was produced with the Statechart real-time code generator STARC.

The first two programs where chosen to test if the presented approach can handle programs with a large number of loops and branches respectively. The Statemate example was used because it consists of automatically generated code and exhibits a code structure which is commonly used in embedded applications (most work is done within a single loop).

6.2.3 Model-Based Code Generation

To test whether the approach is suitable for programs which are developed using model-based design tools, programs generated from MatLab Simulink and SCADE were tested as well. Although the examples were small, it was nevertheless of interest whether their structure could be handled by the partitioning algorithm. To generate input data for the models, code had to be added manually. So their behavior during the run-time measurements was not necessarily typical or realistic.

- Fuelsys: A model of a fuel rate controller created with MatLab Simulink for which C code was generated using the Real-Time Workshop code generator. This model is one of the examples supplied with MatLab and its interface is relatively simple, so input data could be created easily. What is interesting about this example is that the generated code also contained some search and interpolation routines from the code generator. Therefore large portions of this test case are also part of real-world programs.
- Carcontrol: This model was the result of a student project which used the Esterel SCADE Suite to generate code for controlling a Lego Mindstorms car.

6.3 Results

6.3.1 Coverage Requirements

The goal of the first test run was to evaluate how the number of measurement runs per program segment influences the calculated WCET estimates. For this purpose, the test programs were stored in the SPRAM to eliminate any influence of the instruction cache. For two test programs, the WCET was estimated by performing 10, 50 and 250 measurements for each program segment respectively. The results are displayed in

Test	Segment Traces	Segments	Automata	WCET estimate
Carcontrol	10	2	2	1468 cycles
Carcontrol	50	2	2	1481 cycles
Carcontrol	250	2	2	1504 cycles
Fuelsys	10	10	10	8856 cycles
Fuelsys	50	10	10	8943 cycles
Fuelsys	250	10	10	9236 cycles

Table 1: Effects of increasing coverage

table 1. What can be seen immediately is that taking more measurements per program segment results in larger WCET estimates. This observation verifies the assumption that increasing the number of measurements makes it more likely to observe the local worst-case for each basic block and hence makes the WCET estimate more precise. Nonetheless, there is no clear correlation between the number of measurements and the increase of the WCET estimates. Another result of this test run was that performing the measurements requires a considerable amount of time and disk space, even for relatively small examples, as the measurement phase and the timing extraction phase are not parallelized in the current prototype implementation and the file formats are not optimized for efficient storage. That is why all following experiments were conducted with 50 traces per program segment in order to keep the memory requirements for the trace data within reasonable limits.

6.3.2 Cache Behavior Metrics

The effectiveness of the two cache metrics presented in section 4.4.2 were evaluated in a separate test run. Table 2 displays the number of trace automata generated for the different cache metrics and the resulting WCET estimates. The results for the test programs show that the *average distance metric* delivers results which are very close to those of the *maximum distance metric*. As the average distance metric classifies more program segments as similar in terms of their cache behavior, it is able to reuse trace automata more often. Hence a smaller number of measurements must be taken. Although it could be expected that this reuse, which effectively drops some level of context-sensitivity, would affect the WCET estimates negatively, this seems not to be the case. Based on these observations, the average distance metric was adopted for all subsequent experiments.

6.3.3 Comparison of Static and Dynamic Timing Analysis

As the main goal of this work is to develop a precise alternative for static timing analysis, comparing the results of the measurement-based technique with the completely static approach was of particular interest. The experiments were performed with minimal manual intervention. This means that the static analysis was only provided with little additional

Test	Metric	Segments	Automata	WCET estimate
Fuelsys (cached)	average	10	6	14951 cycles
Fuelsys (cached)	maximum	10	10	14675 cycles
Nsichneu (cached)	average	8	5	16182 cycles
Nsichneu (cached)	maximum	8	7	16191 cycles

Table 2: Influence of cache metrics

information about the program structure (e.g. loop bounds), but it was not given any user annotation to improve the precision of the results. For this reason the determined WCET bounds are only rough, but safe approximations of the worst-case execution time. Similarly, due to the small number of traces which were performed for each program segment, the WCET estimates of the measurement-based technique are only a rough approximation of the WCET, too. In contrast to the static approach, these estimates are not safe as the actual WCET value is approximated from below.

The results of the test runs are displayed in figure 15 (note that the scale is logarithmic). Not surprisingly, the measurement-based WCET estimates are considerably smaller than the bounds reported by the static timing analysis. Nevertheless, a closer examination suggests the conclusion that the measurement-based estimates are closer to the actual worst-case execution time than the bounds reported by the static analysis. Manual exploration of the results revealed that the execution time distribution reported by both approaches was identical, i.e. both techniques identified similar worst-case paths through the programs. However, for some program routines with memory accesses the execution times reported by the static analysis were cosiderably larger than the execution times determined by the measurement-based approach. Additionally, Task6 of the DEBIE-1 benchmark has been omitted in figure 15 as no plausible WCET bound could be determined for it within reasonable time.

One example where the completely static timing analysis seems to overestimate the time of data accesses is the random number generation in the DEBIE-1 benchmark. As the benchmark emulates the operating system functions for the original control software, these “random numbers” are read from an array of constant values which are accessed one after another. Some of the analyzed tasks contain a loop of moderate size which repeatedly calls the routine which reads the next value from the array of random numbers. Since the accesses to the respective memory region occur in a very close time frame, it is almost impossible that all of them will result in a cache miss. Nevertheless, this is what the static timing analysis assumes as it is unable to discover that the respective accesses occur to consecutive memory addresses. As the said accesses occur within a loop, the impact of this overestimation is even increased as only the first iteration was considered separately from the remaining iterations during the analysis. Similar effects could be observed when larger parts of memory were copied e.g. in the `memcpy` routine, though the pessimistic assumptions about memory access times might be more realistic here. Despite the obvious overestimation of the completely static approach, the measurement-

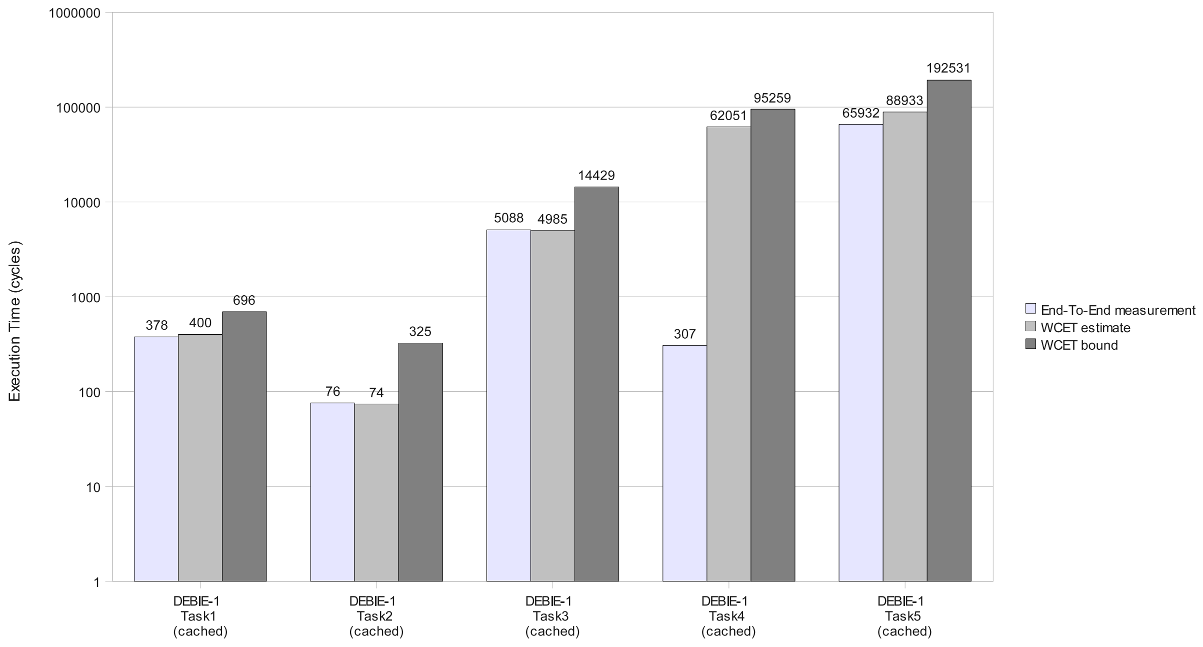

Figure 13: Comparison of context-sensitive and end-to-end measurements

based estimates are probably slightly smaller than the actual worst-case execution time since the number of measurements on which the estimates are based was small. Hence it is unlikely that all local worst-cases were observed during the measurements.

6.3.4 Context-Sensitive and End-To-End Measurements

Additionally, the results of the new approach were compared to the results of simple end-to-end measurements. The outcome of these tests (figure 13) was interesting as it illustrates some problems inherent to all measurement-based methods for timing analysis. Furthermore, some problems of the current implementation of the context-sensitive approach were revealed as well. The general problem of estimating the worst-case execution time of programs based on measurements is that it is unknown under which circumstances the worst-case occurs. Hence it cannot be guaranteed that the worst-case has been covered by any of the measurements. Thus, end-to-end measurements are practically useless for estimating the WCET. This was demonstrated by one of the test cases (DEBIE-1 Task4): though a considerable effort was made for the measurements, the observed end-to-end execution times were considerably smaller than WCET estimates reported by the other approaches. Manual examination of the measurements showed that some routines which were on the WCET path reported by the other approaches were never executed during the end-to-end measurements. However, these routines could be observed in the measurements of the context-sensitive approach, so there are program runs which execute these routines. Hence this test case showed that for programs which rarely execute the routines which are responsible for the worst-case execution, the context-sensitive approach seems to be superior to simpler methods. The program partitioning and the precise control over the measurement runs allow determining the execution time of routines which are not executed very often, as the measurement hardware is able to wait for these program parts before starting the actual trace. Additionally, the context-sensitive approach allows the combination of local worst-case executions which were observed during different measurement runs to obtain an estimate of the global WCET. Nonetheless, the prototype implementation of the context-sensitive approach reported some WCET estimates which were smaller than the maximal execution time observed during the end-to-end measurements. The first reason for this are the delay problems which were already mentioned in section 5.3.2. As a result of the potential delays, the basic blocks at the beginning of some program segments might never be covered completely by the measurements and hence the execution time will be underestimated. Furthermore, insufficient coverage of critical program parts is a potential cause of underestimation. As the number of measurements which were taken during the experiments was relatively small, it is likely that this was another cause for the observed underestimation.

6.3.5 Context-Sensitive and Context-Insensitive Measurements

Finally, WCET estimates which consider the execution context where compared to context-insensitive estimates. For this purpose, the analyzed programs where traced with one single program segment to overcome the delay problems. The annotation phase for the context-sensitive analyses was carried out as in the previous experiments. For the context-insensitive case, the maximal execution time of every basic block was extracted from all of the program traces without consideration of the execution history. This execution time then was annotated to all ICFG nodes which represent the respective basic block. A smaller number of measurements was performed for these experiments than for the previous ones as the focus was not on precisely estimating the WCET (i.e. covering *all* local worst-cases), but on investigating the effect of context information. For this reason, the results presented in figure 14 differ slightly from the previous estimates. Nevertheless, the outcome of the experiments shows that the use of context-information can improve the precision of measurement-based execution time estimates. For two out of three test cases, the context-sensitive approach seems to be able to represent cache effects correctly. Hence, smaller WCET estimates are reported. This effect could not be observed for the smallest of the test cases, probably since the execution time of the program does not benefit from caches due to its linear structure. The results of this comparison suggest that the difference between a context-sensitive and a context-insensitive analysis can be substantial. By increasing the number of measurement runs, this effect can only be intensified, as for every increase in the context-sensitive estimate, the context-sensitive estimate must grow as well. Thus the execution context of execution time measurements should be preserved whenever possible. If this is not done, cache effects cannot be determined correctly, which is why a context-insensitive evaluation might introduce a severe amount of pessimism to the execution time estimates, which renders them less precise.

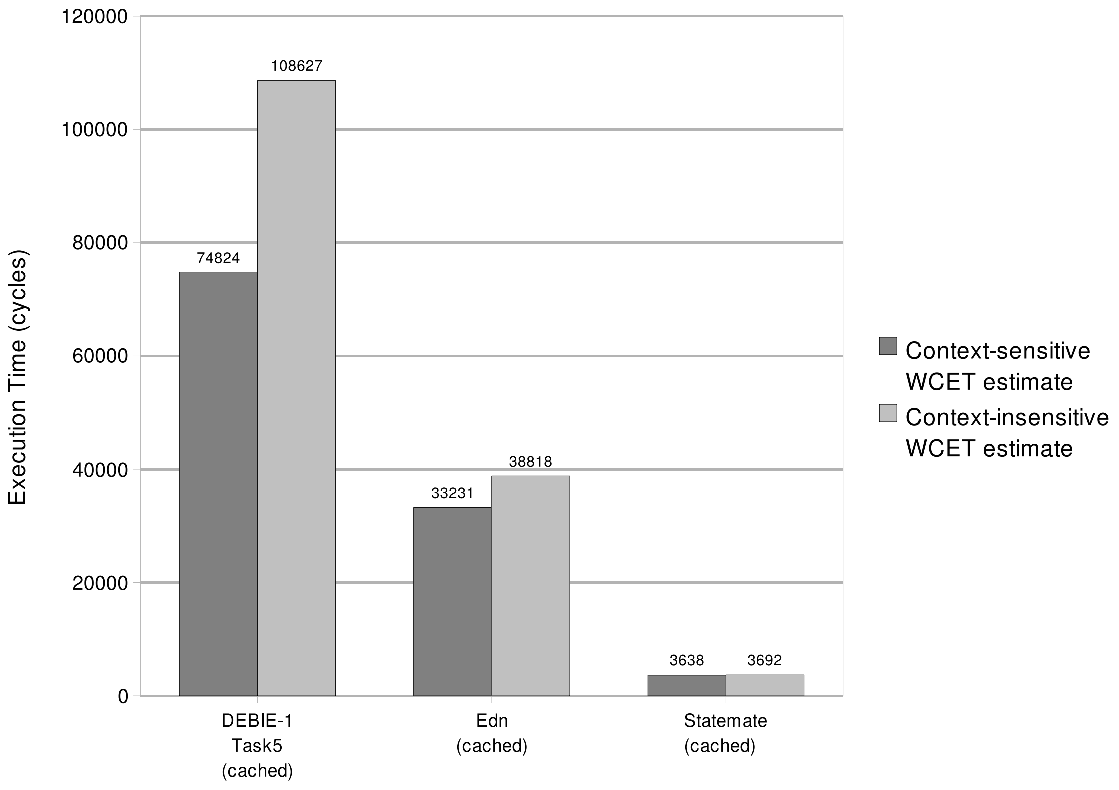

Figure 14: Improvement of WCET estimates through context information

6.4 Evaluation

The key result of the presented experiments is that the approach developed in this thesis works and that the precision of measurement-based WCET estimates can be considerably improved by using context information. Furthermore it has been shown that the method can be applied to real-world programs. Despite some minor problems, this new technique seems to be a good alternative to existing methods for static timing analysis, as long as there is no need for guarantees about the computed WCET estimates. Effectively, this limits the area of application to soft real-time systems. Yet a measurement-based approach might be better suited in this case, especially if the used processor architecture exhibits complex timing behavior or the cost pressure for the final product is very high. The experiments have shown that the technique is able to produce realistic execution time estimates without the intrinsic pessimism of static methods while being more accurate than end-to-end or context-insensitive measurements. Although the new approach seems to be more robust and more precise than existing methods for measurement-based timing analysis, it does not overcome their inherent problems, like the dependence on input data. However, controlling the collection of trace data precisely allows weakening the influence of these problems to the WCET estimate e.g. because it is now possible to enforce measurements within program parts or execution contexts which are executed very rarely. While the precise control of trace data generation makes it more likely that local worst-case executions can be observed, the use of context information allows the precise combination of partial execution times. This makes the calculated WCET estimates less pessimistic. The results of the experiments also show that only measuring each basic block often enough, which is the prevailing paradigm for measurement-based timing analysis, is not enough to determine precise execution time estimates as the execution history might have a significant influence on them. Even though it is in principle possible that the reported context-sensitive WCET estimates are larger than the WCET bounds determined by a static analysis, for example due to an insufficient level of context-sensitivity, the results of the experiments suggest that the WCET estimates are always below the upper timing bound. However, it cannot be guaranteed that the estimate is larger than the WCET, so their might be an underestimation. Nonetheless, based on the experiments, the measurement-based estimates seem to be closer to the WCET than the WCET bounds from a static analysis. Hence the approximation is more precise, but potentially unsafe.

The outcome of the experiments suggests several improvements of the implementation of the algorithms, but it has been proven that the general concepts work very well and can be applied to programs of realistic size. As a first improvement, a closer coupling of the measurement and annotation phase is desirable. By parallelizing the measurements and their processing, the storage requirements for the trace data could be reduced. This should also speed up the analysis time compared to the two-step design of the prototype implementation which first generates the trace data and then extracts the basic block execution times. Additionally, collecting trace data should be guided by previous program runs to enforce certain levels of coverage. Currently, each program segment is measured a fixed number of times and each measurement must be completed within a time bound determined by the user. If a measurement attempt does not produce any data, e.g. because the activation conditions for starting the trace are not met within the given time frame, this has no consequences. Instead, the implementation could be extended to reattempt these measurements until certain coverage criteria are fulfilled. To avoid timeouts during these reattempts, it would be possible to gradually increase the time frame for measurements which did not finish in time during a previous attempt. To increase the coverage of the measurements even further, the process of program partitioning could be adopted to enforce tracing of all branches of a conditional, at least to some extent. These modifications are likely to overcome the problems which were encountered during the experiments.

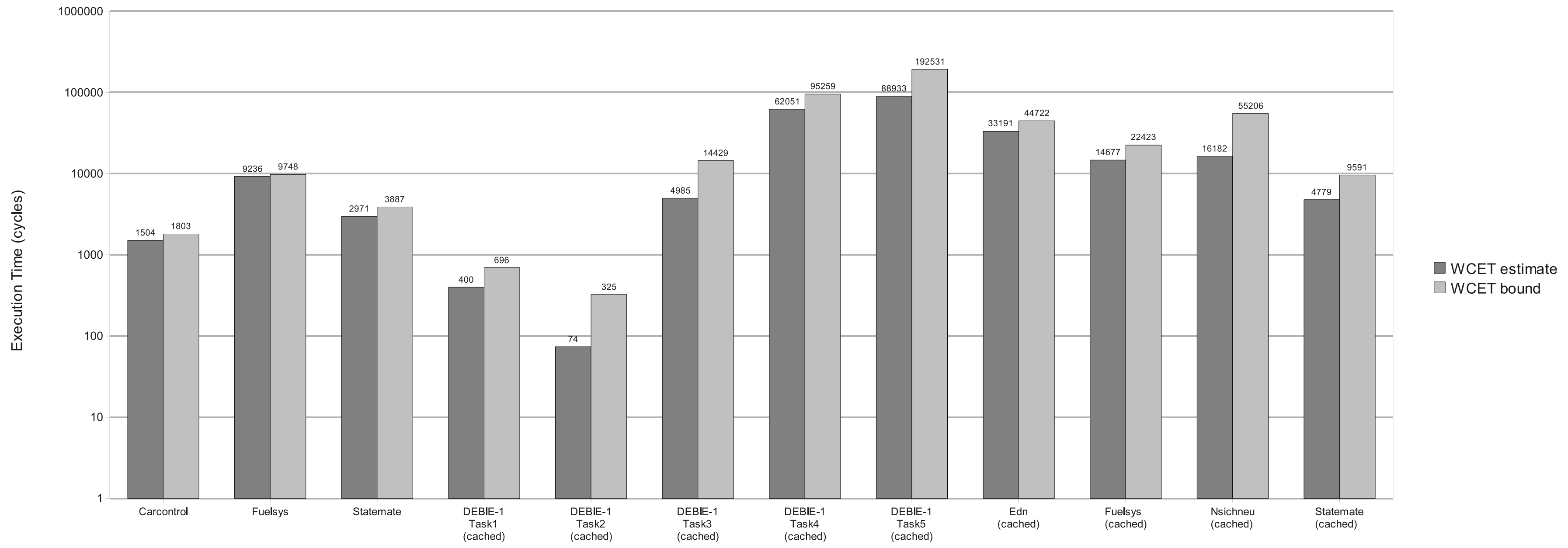

Figure 15: Results from measurement-based and a completely static timing analysis

7 Conclusion

7.1 Summary

This work described a new approach to measurement-based timing analysis which makes use of techniques from static program analysis. In order to generate cycle-accurate program traces, the program to be analyzed is partitioned into program segments which can be measured completely with the trace buffer memory available on the measurement hardware. For each segment, a trace automaton is created to control the collection of trace data during a measurement run. This allows considering a single program segment for generating trace data, while all other segments are ignored for the respective measurement run. Hence context-sensitive execution time measurements are possible despite the limited memory for storing trace data. By incorporating results of a cache analysis, the number of necessary measurements can be reduced by joining program segments which exhibit similar cache behavior.

The technique was implemented within a proven framework for static program analyses which was combined with a widely used software debugger. Experiments with the prototype implementation provided highly encouraging results and suggest incorporation of the analysis and the measurement phases. The results obtained during the experiments show that state-of-the-art measurement hardware can be used to determine WCET estimates of a program automatically, while the precision of the estimates is increased by using context information. The automation of the process does not require the user to provide a lot of information about the analyzed program, but the measurements should be performed in an environment which is typical or realistic for the later use of the software. In order to get good results, a large number of measurements must be performed since the method relies on the assumption that the local worst-case for each basic block of the program was observed during the measurements.

The precise control of measurement runs and their accurate combination is what sets the approach presented in this thesis apart from all other measurement-based methods for timing analysis. The integration of results from static program analyses allows improving its efficiency. Therefore this new method is able to combine the advantages of static and dynamic techniques for timing analysis while overcoming some of their disadvantages. Hence it is a valuable alternative to all existing methods for timing analysis.

7.2 Outlook

While the methods presented in this work have been shown to be applicable to real-world programs, several changes could be made to improve the results or extend the field of application: To make the generation of trace data more efficient, the cache behavior metrics could be modified or extended to consider additional information. For example, accesses to data memory could be factored in without doing a full-scale data cache analysis by using value analysis results. Even if this information is not very precise, it might still suffice to determine the type of memory an access goes to. In effect this would make it possible to distinguish between data accesses which go to fast or to slow memory areas as well as between accesses to cached or uncached areas. Hence it would be possible to determine whether the time of data accesses is likely to vary in different execution contexts. Furthermore, the information about data accesses could be used to determine which program segments are likely to be involved in a worst-case execution and hence should be measured more often than program parts where this is not the case.

A more fine-grained control over the trace data generation would be helpful to improve the quality of the WCET estimates. Though the currently available versions of the MCDS already allow the definition of sophisticated conditions to start a program trace, the number of events which can be combined to form such conditions are limited. These restrictions also bound the number of execution contexts which can be distinguished during the measurements. Apart from adding additional programmable trigger conditions to it, the MCDS could also be improved for the purpose of WCET estimation by simply adding additional trace buffer memory.

Instead of using real hardware to generate traces, a processor simulator or a model synthesized from a specification in a hardware description language could be used as well. While processor simulators are often simplified (e.g. because they do not simulate caches), they might still be useful to estimate the execution time of software during early development phases. In this case, the focus of the analysis would be to identify potential bottlenecks in the application code instead of determining the WCET of the program. On the other hand, processor models derived from synthesizable descriptions should be able to create accurate traces which can be used for WCET estimation. Although this probably requires some manual modifications of the models, the effort should be considerably smaller than for creating an abstract processor model for a completely static approach. A model synthesized from a hardware description language might also be adapted to allow a modification of the overall system state, e.g. by clearing the cache. So, generating a large number of context-sensitive traces from such a model, combined with modifications of the system state, but without modifications of the analyzed program, might be able to generate precise estimates of the worst-case execution time, even for systems with complex cache designs.

The trace automata concept could be adopted to detect other context-sensitive properties of a program as well. Instead of execution times, the traces could be used to detect the maximal number of iteration for loops or the maximal recursion depth of functions. Furthermore, it would also be possible to use trace data for control flow reconstruction to detect the potential targets of computed call or computed jump instructions.

Finally, in contrast to other approaches for determining the worst-case execution time of a program, the technique described in this thesis can be easily adapted to estimate the best-case execution time of a program as well. This can be achieved by modifying the timing extraction phase to extract the minimal execution time of a basic block from the traces instead of searching for the maximum. Additionally, the implicit path enumeration phase would have to be altered to determine a path with the minimal sum of basic block execution times. Conducting these changes should require very little effort, but would increase the area of application for the technique considerably.

Acknowledgements

First of all, I would like to thank Reinhard Wilhelm, Florian Martin and Christian Ferdinand for supporting this thesis. Furthermore, I thank the whole staff of AbsInt Angewandte Informatik GmbH for their assistance during the realization of this work, especially Christoph Cullmann and Gernot Gebhard, who provided a cache analysis library which served as the foundation of the cache behavior analysis implementation, Frank Fontaine, who implemented parts of the trace conversion, and Ingmar Stein, who always willingly shared his knowledge about the TriCore architecture. Additionally, I would like to thank Niklas Holsti for providing the DEBIE-1 benchmark, Lili Tan for supplying the MatLab Fuel System example and Vanessa Hauck for proof-reading.

A Partitioning Algorithms

This part contains the algorithms which have been omitted during the description of program partitioning in section 4.2.3. Some special cases, mainly involving recursive routines, have not been included to make the algorithms more readable.

Algorithm A.1 Calculate the length of the longest path between two nodes in the CFG.

```
    longestLocalPath ($s$, $e$, $m$, $\mathbf{E}$)
        $s$ : start node
        $e$ : end node
        $m$ : mapping $\mathbf{V} \to \mathbb{N}$ for storing partial results
        $\mathbf{E}$ : edges of the CFG

 1: $outgoing \leftarrow \{n \mid (s,n) \in \mathbf{E}\}$
 2: $size \leftarrow |\{i \mid i\ is\ an\ instruction\ in\ basic\ block\ s\}|$
 3: $longest \leftarrow 0$

 4: if $m[s]$ is defined then
 5:     return $m[s]$ // if node has been visited before, use the already calculated result
 6: else if $s = e$ then
 7:     $m[s] \leftarrow size$
 8:     return $size$
 9: end if

10: for all $n \in outgoing$ do
11:     $path \leftarrow longestLocalPath(n, e, m, \mathbf{E})$
12:     $longest \leftarrow max\{longest, path\}$
13: end for

14: $m[s] \leftarrow longest + size$                    // store result and mark node as visited
15: return $m[s]$
```

Algorithm A.2 Calculate the length of the longest interprocedural path between nodes in the ICFG and determine context-sensitive recursion bounds.

```
longestPath ($s$, $e$, $m$, $\hat{\mathbf{E}}$, $call$, $iter$)
    $s$ : start node
    $e$ : end node
    $m$ : mapping $\hat{\mathbf{V}} \to \mathbb{N}$ for storing partial results
    $\hat{\mathbf{E}}$ : edges of the ICFG
    $call$ : mapping $\mathbf{R} \times \mathbf{P} \to \mathbb{N}$ which contains the length of a routine call per context
    $iter$ : mapping $\mathbf{R} \times \mathbf{P} \to \mathbb{N}$ for context-sensitive recursion bounds of routines

 1: $(s_v, s_p) \leftarrow s$        // extract context information from ICFG node
 2: $outgoing \leftarrow \{n \mid (s, n) \in \hat{\mathbf{E}}\}$
 3: $size \leftarrow |\{i \mid i \text{ is an instruction in basic block } s_v\}|$
 4: $longest \leftarrow 0$

 5: if $m[s]$ is defined then
 6:     return $m[s]$  // if node has been visited before, use the already calculated result
 7: else if $s = e$ then
 8:     $m[s] \leftarrow size$
 9:     return $size$
10: end if

11: for all $(n_v, n_p) \in outgoing$ do
12:     if $routine(s_v) \neq routine(n_v)$ then
13:         $path \leftarrow call[(routine(n_v), n_p)] * iter[(routine(n_v), n_p)]$
14:     else if $s_p \neq n_p$ then
15:         // same routine, but different context $\Longrightarrow$ recursive call
16:         $iter[(routine(n_v), n_p)] \leftarrow iter[(routine(s_v), s_p)] - 1$  // update recursion bounds
17:         $path \leftarrow call[(routine(n_v), n_p)] * iter[(routine(n_v), n_p)]$
18:     else if $routine(n_v)$ is not recursive then
19:         $path \leftarrow longestPath((n_v, n_p), e, m, \hat{\mathbf{E}}, call, iter)$
20:     else
21:         $path \leftarrow 0$        // ignore calls to the same routine and the same context
22:     end if
23:     $longest \leftarrow max\{longest, path\}$
24: end for

25: $m[s] \leftarrow longest + size$        // store result and mark node as visited
26: return $m[s]$
```

Algorithm A.3 Calculate the maximal length of all interprocedural paths in the ICFG by constructing a mapping $call \in \mathbf{R} \times \mathbf{P} \rightarrow \mathbb{N}$ which returns the maximal length of the intraprocedural path from the start to the end node for every routine in every valid context. Also construct a mapping $iter \in \mathbf{R} \times \mathbf{P} \rightarrow \mathbb{N}$ for context-sensitive recursion bounds.

```
longestRoutinePaths ($\mathbf{R}$, $\hat{\mathbf{E}}$)
   $\mathbf{R}$ : the routines contained in the ICFG
   $\hat{\mathbf{E}}$ : edges of the ICFG

 1: for all $(r,p) \in \{(r,p) \mid r \in \mathbf{R}, p \in contexts(r)\}$ do
 2:    $m \leftarrow$ empty marking
 3:    $\mathbf{E}_r \leftarrow \{(s,t) \mid \exists p \in contexts(r): \ ((s,p),(t,p)) \in \hat{\mathbf{E}}$
                  $\wedge\ routine(s) = r \ \wedge\ routine(t) = r\}$
 4:    $call[(r,p)] \leftarrow longestLocalPath(start(r), end(r), m, \mathbf{E}_r)$
 5:    $iter[(r,p)] \leftarrow iterations(r)$
 6: end for

 7: $workList \leftarrow \{(r,p) \mid r \in \mathbf{R}, p \in contexts(r)\}$
 8: repeat
 9:    $(r,p) \leftarrow$ pick element of $workList$
10:    $oldResult \leftarrow call[(r,p)]$
11:    $call[(r,p)] \leftarrow longestPath(start(r), end(r), m, \hat{\mathbf{E}}, call, iter)$

12:    if $oldResult \neq call[(r,p)]$ then
13:       for all $(r',p') \in \{(r',p') \mid \exists((s,p'),(t,p)) \in \hat{\mathbf{E}}: \ routine(s) = r' \wedge routine(t) = r\}$
          do
14:          $workList \leftarrow workList \cup \{(r',p')\}$          // insert callers of $r$ into work list
15:       end for
16:    end if

17:    $workList \leftarrow workList \setminus \{(r,p)\}$
18: until $workList = \emptyset$

19: return $(call, iter)$
```

Algorithm A.4 Calculate recursion bounds and the length of interprocedural paths before partitioning the ICFG into program segments.

```
partitionICFG ($\mathbf{R}$, $\hat{\mathbf{E}}$, $s$)
    $entry$ : entry node of the ICFG
    $l$ : maximal length of a path ICFG
    $\mathbf{R}$ : the routines contained in the ICFG
    $\hat{\mathbf{E}}$ : edges of the ICFG

1: $(entry_v, entry_p) \leftarrow entry$
2: $r \leftarrow routine(entry_v)$
3: $exit_v \leftarrow exit(r)$
4: $m \leftarrow$ empty marking

5: $(call, iter) = longestRoutinePaths(\mathbf{R}, \hat{\mathbf{E}})$
6: $partition(entry, (exit_v, entry_p), m, \hat{\mathbf{E}}, l, call, iter)$
```

Algorithm A.5 Find nodes that have to be visited on every path between two nodes in the ICFG. Since the start and the end node are known, this problem is less complex than the general search for dominators of basic blocks.

```
findDominator ($s$, $e$, $m$, $\hat{\mathbf{E}}$, $call$, $iter$)
    $s$ : start node
    $e$ : end node

1: $dominators \leftarrow \bigcap_{p \in s \to e} \{n \mid n \in \hat{\mathbf{V}},\ \exists k \in \mathbb{N} :\ p^k = n\}$        // nodes on every path

2: if $dominators = \{s, e\}$ then
3:     abort
4: end if

5: $middle \leftarrow$ pick element from $dominators$ with maximal address distance to $s$ and $e$
6: return $middle$
```

References

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation*, pages 85–96. ACM, 1997.

[aiT] AbsInt aiT WCET Analyzer. `http://www.absint.com/ait/`.

[BB00] Guillem Bernat and Alan Burns. An Approach to Symbolic Worst-Case Execution Time Analysis. In *25th IFAC Workshop on Real-Time Programming*, 2000.

[BB06] Adam Betts and Guillem Bernat. Tree-Based WCET Analysis on Instrumentation Point Graphs. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 558–565. IEEE Computer Society, 2006.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 279–288, 2002.

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

[BL92] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70. ACM, 1992.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[DP07] Jean-François Deverge and Isabelle Puaut. Safe Measurement-Based WCET Estimation. In Reinhard Wilhelm, editor, *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Fer97] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.

[FH04] Christian Ferdinand and Reinhold Heckmann. aiT: Worst-Case Execution Time Prediction by Static Programm Analysis. In Ren Jacquart, editor, *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, pages 377–384. Kluwer, Boston, Mass., 2004.

[FMW97] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCTRTS '97) on June 15-18, 1997 at Las Vegas, Nevada*, pages 37–46. ACM Press, 1997.

[HGB$^+$08] Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET 2008 – Report from the Tool Challenge 2008 – 8th intl. Workshop on Worst-Case Execution time (WCET) analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3.

[IPea] Infineon Multi-Core Debug Solution (MCDS). `http://www.ip-extreme.com/IP/mcds.html`, as of July 13, 2009.

[IPeb] Infineon Multi-Core Debug Solution (MCDS) Brochure. `http://www.ip-extreme.com/downloads/MCDS_brochure_080128.pdf`, as of July 13, 2009.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 456–461, New York, NY, USA, 1995. ACM.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories.* PhD thesis, Chalmers University of Technology, 2002.

[Mar99] Florian Martin. *Generating Program Analyzers.* PhD thesis, Universität des Saarlandes, 1999.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC '98), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS) on March 28-April 4, 1998 at Lisboa, Portugal*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94, Berlin, 1998. Springer.

[MB08] Amine Marref and Guillem Bernat. Towards Predicated WCET Analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Mea55] George H. Mealy. *A Method for Synthesizing Sequential Circuits.* Bell Systems Technical Journal 34 (5), 1955.

[MH08] Albrecht Mayer and Frank Hellwig. System Performance Optimization Methodology for Infineon's 32-bit Automotive Microcontroller Architecture. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 962–966. ACM, 2008.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer, 1999.

[PSM09] Markus Pister, Marc Schlickling, and Mohamed Abdel Maksoud. Semi-automatic Derivation of Abstract Processor Models. Reports of ES_PASS, ES_PASS, June 2009.

[RGBW06] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Predictability of Cache Replacement Policies. Technical report, Sonderforschungsbereich / Transregio 14 on Automatic Verification and Analysis of Complex Systems (AVACS), 2006.

[Ros03] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill, fifth edition, 2003.

[RWT$^+$06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[Seh05] Daniel Sehlberg. Static WCET Analysis of Task-Oriented Code for Construction Vehicles. Master's thesis, October 2005.

[Sha89] Alan Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15:875–889, 1989.

[SP07] Marc Schlickling and Markus Pister. A Framework for Static Analysis of VHDL Code. In Christine Rochange, editor, *Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis at Pisa, Italy*, 2007.

[SSPH06] Stefan Schaefer, Bernhard Scholz, Stefan M. Petters, and Gernot Heiser. Static Analysis Support for Measurement-Based WCET Analysis. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Work-in-Progress Session*, 2006.

[Tec07a] Infineon Technologies. TC1767/97ED Emulation Devices Target Specification, v1.4, December 2007.

[Tec07b] Infineon Technologies. TC179 32-Bit Single-Chip Microcontroller Target Specification, v1.5, November 2007.

[The03] Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis. Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis.* PhD thesis, Saarland University, 2003.

[UDE] pls Development Tools Universal Debug Engine (UDE). `http://www.pls-mc.com`.

[WCC] WCET Tool Challenge 2008 Wiki . `http://www.mrtc.mdh.se/projects/WCC08/doku.php`, as of August 6, 2009.

[WCE] Mälardalen WCET Benchmark Suite . `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, as of August 6, 2009.

[WE01] Fabian Wolf and Rolf Ernst. Execution Cost Interval Refinement in Static Software Analysis. *Journal of Systems Architecture*, 47(3-4):339–356, 2001.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem — Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[WEY01] Fabian Wolf, Rolf Ernst, and Wei Ye. Path Clustering in Software Timing Analysis. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(6):773–782, 2001.

[Wil05] Reinhard Wilhelm. Determining Bounds on Execution Times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.

[WKRP08] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Timing Analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Oct. 2008.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Proc. Conference on Design, Automation, and Test in Europe*, Mar. 2005.