FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Julius Wiedmann

Implementation and Evaluation of Trace-Based Timing Analysis

Bachelorarbeit im Fach Informatik

2. Dezember 2019

Please cite as:
Julius Wiedmann, “Implementation and Evaluation of Trace-Based Timing Analysis”, Bachelor’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, December 2019.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Julius Wiedmann)
Erlangen, 2. Dezember 2019

ABSTRACT

Many embedded software tasks require for their correct scheduling in real-time systems knowledge of their worst-case execution time (WCET). Eventually, this upper bound is calculated by static analysis tools that rely on precise hardware models of the target architecture. Since modern micro-processors often contain undocumented or unpredictable components, these static analysis methods are for these kinds of processors overly pessimistic, leading to a demand for new analysis approaches. This thesis depicts the implementation and evaluation of a hybrid WCET analysis approach based on non-intrusive instruction-level trace measurement. By doing so, it shows the general benefits of the hybrid analysis compared to a static analysis, the risk of underestimation due to cache effects as well as the influence of statement and path coverage on the accuracy.

KURZFASSUNG

Oftmals basiert die korrekte Ausführung von eingebetter Softeware auf der Definition ihrer längsten Ausführungszeit (WCET). Diese Obergrenze wird in vielen Fällen von statischen Analysewerkzeugen berechnet, welche ein präzises Hardwaremodell der Zielarchitektur benötigen. Da allerdings neuere Prozessoren oftmals undokumentierte oder unvorhersehbare Komponenten enthalten, sind diese statischen Analysemethoden für diese Arten von Prozessoren übermäßig pessimistisch, was zu einem Bedarf an neuen Herangehensweisen führt. Diese Arbeit erläutert die Implementierung und Evaluation einer hybriden WCET Anaylse basierend auf berührungsfreien Ausführungsmessung auf Instruktionsebene. Dadurch wird sowohl der Vorteil der hybriden gegenüber der statischen Analyse, die Gefahr ihrer Unterabschätzung durch Cache-Effekte sowie der Einfluss von Anweisungs- und Pfadüberdeckung auf den Genauigkeitsgrad gezeigt.

CONTENTS

1 INTRODUCTION

With the recent increase of embedded systems in areas like automotive electronics or avionics, a demand for tools has arisen that analyze the timing behavior of the tasks executed on these so called real-time systems. Otherwise, a miscalculated timing behavior could cause in embedded systems an entire system failure. Since such a failure in a car or an airplane can lead to heavy danger for human life, industries like these are often subject to safety standards that require reliable upper bounds for the longest possible execution time for each performing task, like for example the certification processes described at the DO-178C [10] in commercial aircraft software.

One way to estimate these upper bounds is through static analysis. This approach relies on abstraction of the performing task and a hardware model of this real-time system. However, the recent rise of hardware complexity and the fact that some processors contain undocumented parts makes this approach increasingly difficult [7].

One approach of providing an analysis for these undocumented respectively unpredictable processors is through exhaustive end-to-end measurement of the task for different inputs. Since its upper bound estimations lack in reliability, this approach cannot be the solution for safety critical systems.

The idea of designing a hybrid analysis by combining elements from the static analysis with measurements has been applied in a huge amount of research papers in the last years [7, 4, 3, 14, 5]. Their goal is to evaluate the possibility whether hybrid analyses could close the gap for the kind of hardware on which the static analysis approaches fails and an end-to-end measurement-based analysis could not provide with the needed safety.

The motivation for this thesis is to implement a hybrid analysis based on the static analysis tool platin [6]. Its objective is to design an approach combining design choices of previous papers like the non-intrusive measurement execution [7] with so far untested elements like the measurement aggregation on instruction level or the trace extraction with a cycle-accurate chip simulator.

The underlying goal is to answer the question whether the effects of measuring executions on instruction level improve or deteriorate the overall estimation performance of the analysis. Furthermore, the hybrid analysis results are discussed in the light of a possible underestimation due to unmeasured cache effects.

2 FUNDAMENTALS

This chapter gives a short introduction on the field of worst-case execution time (WCET). The first section gives an insight into real-time systems and why it is necessary to define their longest possible execution time. The following sections discuss established analysis methods on this field. In that process, it is the goal of this chapter to point out the problems of these conventional approaches and to show the necessity for the design of new methods.

2.1 WCET Analysis in General

Before understanding worst-case execution time analysis methods and their concept, one has to understand at first the underlying problem. Therefore, the following section gives a short introduction why real-time systems need to be analyzed with regard to their longest execution time.

2.1.1 Real-Time Systems

The special property of real-time systems is that their correctness does encompass more than the logical correct calculation of the output. In addition, this output must be delivered in a certain predefined response-time. Failing to meet this time constraint can lead to various consequences ranging from performance decrease to complete system failure. Depending on the impact of these consequences real-time systems can be classified in three categories.

- Hard real-time systems
- Firm real-time systems
- Soft real-time systems

In *hard real-time systems* the response-time has to be satisfied at all costs. Otherwise it would lead to the consequences of a system failure. In *firm real-time systems* the computation process has also hard deadlines, but a missing of these deadlines can be tolerated by the system. If a violation of a timing constraint only leads to a performance decrease of the systems, it is called a *soft real-time system* [2]. In order to guarantee the safety of the real-time systems, it is therefore necessary for the developer to not only test a task on its logical correctness but also to detect the longest possible execution-time this task could have.

2.1.2 The Worst-Case Execution Time

Generally, the execution time of a real-time task varies for each input data. The dark bars in Figure 2.1, for example, depict the time distribution of all possible execution of an exemplary program. The shortest possible execution on the left is called *best-case execution time* (BCET) and the longest possible execution time on the right is the so called *worst-case execution time* (WCET). The whites bars below depict the time distribution of the already measured executions. It is clear that the WCET cannot be attained with full certainty without measuring the execution times for all possible inputs. A program with only one 32 Bit Integer as input, for example, would have $2^{32} = 4.294.967.296$ different input variations. Due to this combinatorial explosion it is impossible to measure the program for all of these inputs which leads to the demand of a more structural method for the analysis of the WCET.

2.1.3 The Worst-Case Execution Time Analysis

In practice, there are two approaches of WCET analysis. One method relies on exhaustive end-to-end measurement of all input variations, the so called *measurement-based analysis*. The other way of WCET analysis uses the implicit path enumeration technique (IPET) in order to analyze the programs structure. It is called a *static analysis*. The main criteria for the evaluation of these analysis methods are *precision* and *safety*. Precision means the ability of the analysis to produce upper bounds that are close to the actual WCET. Safety describes the confidence that the actual WCET is not higher that the WCET upper bound [15]. The analysis's ability to produce safe and precise upper bounds is called *estimation performance* in this thesis. As the following sections show, the main difference of these two WCET analysis approaches lies not only in their different implementation but more importantly in the tradeoff between the criteria safety and precision.

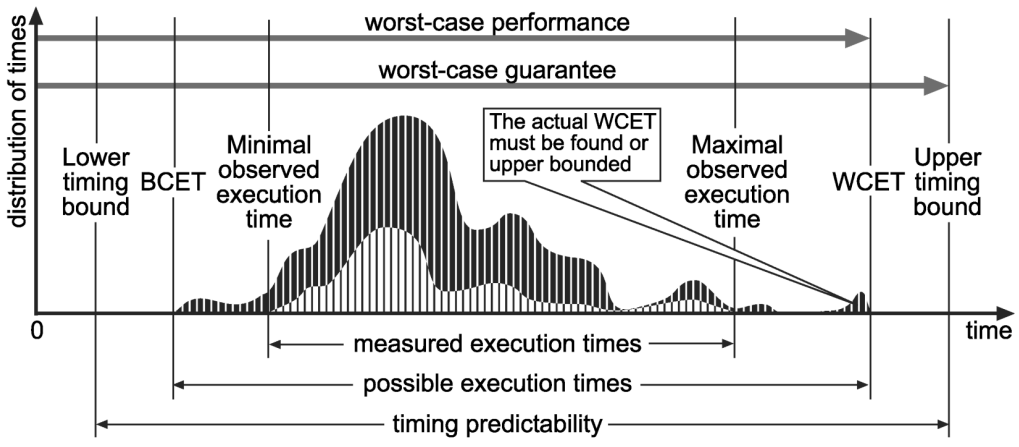

Figure 2.1 – The black bars show the actual execution time distribution of an exemplary real-time task. The white bars below show the distribution of the execution times measured while testing. The figure shows therefore the problem of estimating the WCET without knowledge about the execution time for unmeasured input data. The figure is taken from [15].

2.2 Measurement-based Analysis

One way of WCET estimation is the simple approach of exhaustive end-to-end measurements of the program. A concept closely linked to that approach involves the metrics in order to evaluate the measurement progress.

2.2.1 General Approach

The general approach of gaining knowledge about the execution time distribution shown in Figure 2.1 is end-to-end measurement, a method commonly used in the industry [15]. The goal is to measure the execution time of the whole task for as many different inputs as possible and therefore approximate the WCET. The longest execution time that occurs while measuring is called worst-observed execution time (WOET).

2.2.2 Coverage Metrics

Since it is practically impossible to cover all input variations in programs with higher complexity, there is a set of coverage metrics, to evaluate the possibility that the actual WCET might be still higher that the WOET. Two very important ones are *code coverage* (in this thesis called *statement coverage*) and *path coverage*. Statement coverage describes the proportion of the statements that are covered in the measurement process in relation to the set of statements of the task in total. Path coverage calculates the share of the covered paths in relation to all possible paths [11]. It is important to stress the point that a full statement coverage does not imply a full path coverage. In the program of Listing 2.1 two of the paths are enough to cover all statements, for example one path covering the two then-blocks and one paths covering the two else-blocks leaving the then-else respectively else-then combination uncovered.

```
1  void f(int input){
2    if(input % 3 == 0){
3      /* then-block */
4    }else{
5      /* else-block */
6    }
7    if(input % 2 == 0){
8      /* then-block */
9    }else{
10     /* else-block */
11   }
12 }
```

Listing 2.1 – The structure of the consecutive if-else statements enables four different paths through the program. Yet two of them are enough to reach full statement coverage.

2.2.3 Advantages and Disadvantages

The advantage of the end-to-end measurement is its simple design and implementation. On the other hand it does not produce safe upper bounds because executing the program for all input variations is impossible. The analysis method of the following section solves this problem by using the IPET in order to reduce its complexity.

2.3 Static Analysis

This sections explains the implementation of a static analysis based on the approach of PLATIN [6]. It shows not only the functionality and purpose of its two main steps but also its overall strengths and weaknesses.

2.3.1 Representation of the Program Flow in a Control Flow Graph

The basis for the analysis is the initial representation of the input program. This can be for example the source code as well as the binary code. In PLATIN it is the program's binary and a compiler generated representation of the programs control flow. The first task of computing the WCET is to create a representation of the programs control flow. One way of doing this, is the construction of a control-flow graph (CFG). An exemplary CFG is displayed in Figure 2.2. It shows a program, which contains at first an if-else statement, followed by a loop. While the nodes in the graph represent the basic blocks of the program, the directed edges show the possible ways a specific execution can take trough the program. A *basic-block* is a sequence of instructions on object code level, that contains no branch instructions except on its end. A possible execution would be for example the evaluation of the condition as true, the execution of the then-block and 5 iterations in the following loop. The goal of this first step of the computation is to assign an execution time to each of these basic blocks, by adding up the execution times of each single instruction of this block. In a static analysis this means, that for every instruction its longest possible execution time must be calculated. An other challenge of static analysis is the correct estimation of the execution time of a loop. So, the programmer must provide additional information about a theoretical upper bound for loop iterations, the so called *flow facts*.

2.3.2 The Difficulties of Implementing an accurate Hardware Model

Since there is a huge influence of underlying hardware on the execution overall time, it is a complex task to estimate a safe and precise worst-case execution time for each instruction. A very important influence of the hardware is for sure the effect of caching since cache hits and cache misses produce very different execution times. A cache hit means that the accessed data is already in the cache and a cache miss means that the accessed data is not in the cache and has to be fetched from the RAM. Since the hardware model has to be implemented for every CPU platform PLATIN does not provide a modeling of the cache state for every architecture. For the instruction set architecture (ISA) used in this thesis, for example, it always calculates with the most pessimistic outcome in a situation of uncertainty. It is this pessimism due to insufficient modeling of hardware effects like pipelining or caching in the target architecture that makes the application of static analysis for processors with undocumented or unpredictable paths increasingly difficult [7]. Further, the complexity of these hardware make it hard to construct a correct model of it what eventually leads in a underestimation through static analysis [12].

2.3.3 Using the Implicit Path Enumeration Technique in order to reduce Complexity

Given that there is a GFG and each of the blocks has its execution time assigned to it, the task of the second step in a static analysis is to transform the problem of finding the most expensive path though this CFG into an equivalent integer linear progam (ILP) optimization problem, described in

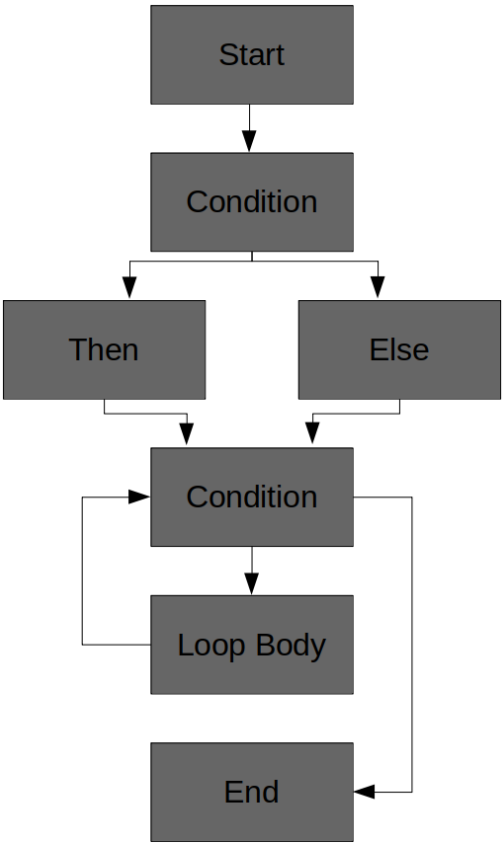

Figure 2.2 – The CFG of this exemplary program contains an if-else statement followed by a loop.

the following way. Given that there are $n \in \mathbb{N}$ edges and $a_0, a_1, \ldots, a_{n-1} \in \mathbb{N}$ edge weights, find a set of $x_0, x_1, \ldots, x_{n-1} \in \mathbb{N}$, representing the execution frequency of the edges, in order to maximize

$$a_0 x_0 + a_2 x_2 + \ldots + a_{n-1} x_{n-1} \forall x_i \in \mathbb{N}, 0 \leq i < n$$

By solving this optimization problem the values of $x_0, x_1, \ldots, x_{n-1}$ represent these execution frequencies in the CFG, which together form the most expensive path through the program. In addition, there is also a mathematical expression containing constraints on $x_0, x_1, \ldots, x_{n-1}$ and only if a instance of $x_0, x_1, \ldots, x_{n-1}$ represents a feasible path this expression is evaluated as true. For example, given that x_j would represent the transition from the if-else-condition block to the *then*-block in the program of Listing 2.1 and x_k would be the representative for the corresponding *else*-block, $x_j + x_k = 1$ would be a constraint saying that either the *then*-block or the *else*-block is possible but neither both or none of them. The advantage of now using the IPET can be seen by looking at the code snippet of Listing 2.2.

The function `rand()` in this example returns a random number between 0 and 1. Therefore, there are two optional paths inside of each loop iteration leading to a total of 2^{100} paths. The problem of an explicit examination of each path is not only impossible but also most likely not necessary if you consider that `j++` and `k++` have the same timing costs. The main contribution of

```
1 for (i=0; i<100; i++) {
2   if (rand() > 0.5){
3     j++;
4   }else{
5     k++;
6   }
7 }
```

Listing 2.2 – The loop contains 2^{100} different paths making an explicit examination of each path impossible. Taken from [9]

this IPET is therefore to implicitly consider both paths inside the loop as part of the worst-case path instead of explicitly enumerating all 2^{100} variations [9].

2.3.4 Advantages and Disadvantages

In summary, the main advantage of the static analysis is its ability to produce safe upper bounds. But the downside of this safety is often the lack of precision in a case of missing information about the hardware state.

2.4 Conclusion

The most important aspect of these conventional analysis approaches is the tradeoff between safety and precision. Indeed the static analysis is able to construct a safe upper bound for the WCET. But this often happens at cost of precision due to insufficient hardware modeling. This tradeoff between static and measurement-based analysis is the reason why there are many publications in recent years combining design elements of these two analyses in order to gain better results for undocumented/unpredictable processor architectures, as the following chapter shows.

3 RELATED WORK

The previous chapter depicted the static and measurement-based WCET analysis. The approach of taking design elements from the conventional static analysis and combining themt with measurements is called a *hybrid WCET analysis*. Its goal is to provide precise WCET estimations for CPUs whose hardware cannot be modeled and therefore a static analysis is infeasible. In addition, its estimations are expected to be more safe compared to a pure end-to-end measurement as the following section shows.

3.1 Hybrid Analysis in General

The industry-grade tool for hybrid WCET analysis is the TimeWeaver from *AbsInt Angewandte Informatik GmbH* [7]. It conducts a WCET analysis by combining trace information on instruction-level with the static analysis approach of aiT tool chain.

3.1.1 Extraction of Trace Information

The trace information is provided by *embedded trace units*. While the processor executes the given program these units deliver information of the programs control flow in form of *trace segments* and *trace events*. A trace event is an instruction that could change the value of the program counter. The instructions between these trace events are the trace segments. Later in the analysis these trace events are mapped to the trace points in the CFG, the segments to the paths between these points in the CFG. The advantage of this approach is that these *trace snippets* provide execution time information without a theoretical modeling of the processors hardware. Thus, this approach tries to avoid the pessimism of many static approaches.

3.1.2 Analysis Steps

The analysis contains four steps all depicted in Figure 3.1. The first task is the *Decoding Phase*. It is mainly the same as in the static aiT tool. The goal is the construction of a CFG. Then the trace information is gathered in the *Microarchitectural Analysis Phase*. The timing information of these trace snippets are then embedded in the CFG, that is transformed to an ILP in the *Path Analysis Phase* and solved by an ILP solver. In order to make an assessment of the analysis results possible at last a *Visualization Phase* was added.

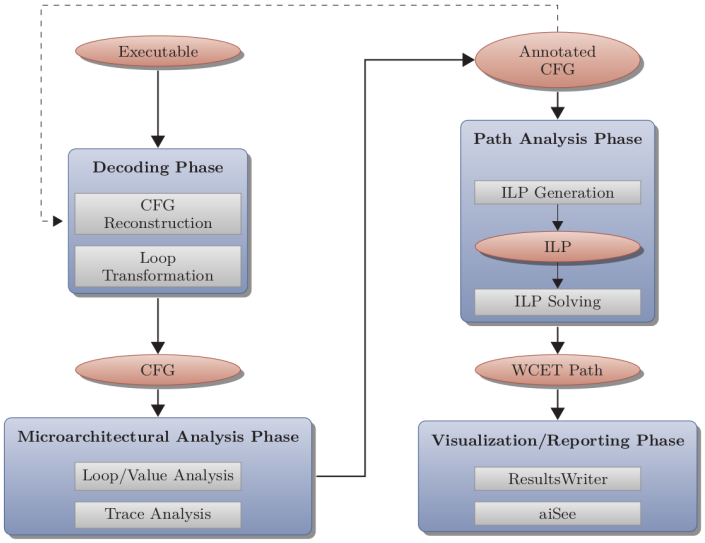

Figure 3.1 – The analysis of the TimeWeaver contains the Decoding Phase, the Microarchitectural Analysis Phase, the Path Analysis Phase and a Visualization Phase.

3.1.3 Evaluation of the Results

Finally, the results of the TimeWeaver are compared to the WOET extracted from the measured traces. By that, the difference between these two WCET estimation is taken in order to evaluate the performance of the hybrid analysis. The results of that can be seen in Figure 3.2. The difference of the hybrid analysis is because of its ability to compose maximum execution times of code segments from multiple trace snippets. Therefore, the hybrid estimation is always a little bit higher than the WOET in the traces. Although the difference of these two estimation performances is very little, the validity of the evaluation itself is questionable. It does not compare the hybrid results with the actual WCET or the upper bound of the static aiT analysis, which is due to the missing of a timing model of the used processor.

3.2 Related Work

In general, the field of WCET estimation has seen a large amount of research in recent years, with many papers proposing a great variety of approaches. While most of the publications chose an analysis based on object code level, there are papers like [14], [3] that analyze the program based on the source code level. Although this approach benefits from the high portability of programming languages like C, the approach in [14] requires additional instrumentation code in order to create the timestamps for the measurement. Since this alteration of the program code changes the execution of the task the consistency of the execution time in measurement and application can not be assured, leading to the so called probe effect. In contrast to that, the TimeWeaver[3] proposes a mechanism that enables a hybrid WCET analysis at source-code level by collecting the timestamps at object-code level instead. Therefore, it can operate without intrusive instrumentation code and in this way without the probe effect.

Application	Trace [cycles]	Estimate [cycles]	Diff [%]
crc	809068	829039	2.47
edn	4788025	4791420	0.07
eratosthenes sieve	368345	369803	0.40
dhrystone	168093	177314	5.49
md5	127857	131718	3.02
nestedDepLoops	2747357	2747359	0.00
sha	23426161	23815350	1.66
Avionics Task	420677	498028	18.38
Automotive Task 1	65058	71964	10.62
Automotive Task 2	27215	28967	6.44
Automotive Task 3	17386	18595	6.95
Automotive Task 4	101749	109302	7.42

Figure 3.2 – The TimeWeaver estimation performance compared to the WOET shows little difference. Taken from [7].

Another difference among many publications is the representation of the programs control flow. B. Dreyer et al. proposed a non-intrusive hybrid WCET analysis with usage of a CFG in [5] and one using a waypoint graph in [4]. The papers showed that a context-sensitive execution time measurement is possible with both methods, context-sensitive meaning in this case the consideration of the hardware state triggered by previous executions and therefore enabling the distinction between different loop iterations. The question of whether the data is gathered offline like in [3] or online like in [5] is another important aspect of execution time measurement. The advantage of the online data aggregation of [5] is that the data that is provided by a field-programmable gate array (FPGA) is continuously fed into the trace extraction module. In an offline data aggregation the raw trace data is at first stored in total thus leading to a large storage usage. [3] stated that their approach produced 160MB per second. In summary, all four approaches use the collection of measurement data on the level of basic blocks respectively the code segments between waypoints.

3.3 Summary of Conventional Methods and Distinction of this Approach

In summary, a common approach of many papers is an object-code level based analysis with offline measured block execution times. A common strategy of avoiding the probe effect is the usage of embedded trace units in processors of FPGA. In contrast to that, the approach presented in this thesis relies on measurements by trace segments of cycle-accurate chip emulators. A second significant difference is the extraction of measurement data on instruction level, as described in the next chapter.

4 APPROACH AND IMPLEMENTATION

This chapter illustrates the general approach and implementation of the used hybrid analysis. The overall goal is not only to show its functionality but in addition its possible weaknesses and strengths.

4.1 Approach

This section describes the design of the hybrid WCET analysis. At first, there is an overview of the general approach. After that, the following three sections relate to the three main parts of the analysis process. Each of these sections gives a general description of the design concept and has a counterpart in Section 4.2, which depicts the concrete implementation of the underlying technique.

4.1.1 The General Analysis Process

In order to study the effects of variations in the implementation in later stages of this thesis, the whole design concept is subject to a fully modular construction. This enables not only a easier modification of the implementation but also a better evaluation of possible interim results. In general the analysis process contains three main parts, of which each of them can be further divided into separate sub-parts. These are

- preparation of the analysis input format
- measurement of the execution time
- computation of the WCET

Figure 4.1 illustrates the process as a whole. The first stage shows the compilation and preparation of the analysis input. The next stage contains the actual program execution and the transferring of the measurement results in a form, which can be embedded in the last phase, the computation of the WCET.

4.1.2 Input Format of the Program

In order to execute the program in the measurement process and to conduct the computation of the WCET, the initial file not only needs to be compiled but also analyzed with regards to its inner structure and program flow. The tasks of this preparation phase are described in the following.

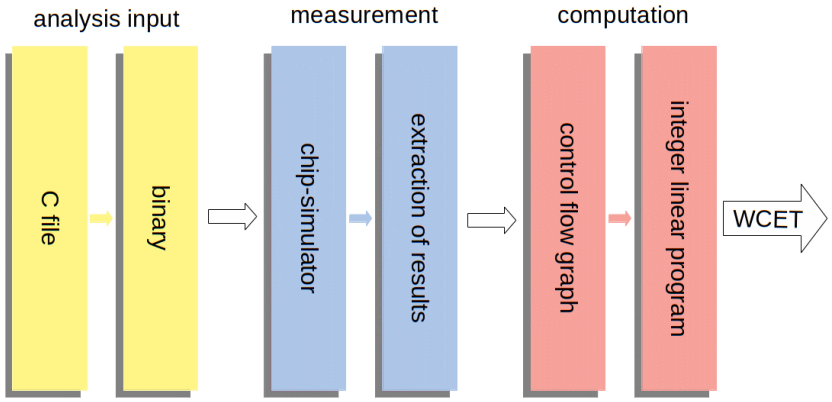

Figure 4.1 – The general analysis process contains the three phases. The first phase prepares the format of the program. The second phase measures the execution time. The last phase computes the WCET.

4.1.2.1 Executable Binary File

In general, different WCET tools work on different representation levels of the program. As described in Section 2.3, these representation levels describe the format, in which the program is passed to the analysis. Since the last stage conducts its computation of the WCET based on object code level, the initial file in C code needs to be compiled and passed to the second as well as to the last stage.

4.1.2.2 Flow Facts

Flow facts are necessary in order to perform a correct computation of the WCET, see Section 2.3. After the compilation they are passed to the computation process through the binary file and are used at the definition of an ILP, that is part of Section 4.1.4.2. Since these flow facts are needed in order to perform a correct computation in the last stage of the analysis, they have to be defined in this approach in the preparation phase manually.

4.1.2.3 Partitioning on Object Code Level

The last task of the first stage is the transformation of the program in a form, which can be used to extract the program flow. This happens through partitioning on object code level. The goal is to divide the program into basic blocks, see Section 2.3. The result of this partitioning serves as the foundation of the construction of the CFG.

4.1.3 Measurement of the Execution Time

The task of this phase is the execution of the previously created binary file and the extraction of the measurement times, which are then passed to the computation phase. These two steps are described in the following two sections.

4.1.3.1 Execution of the Program on a Chip Simulator

In this approach the program is not executed on a real hardware. The execution is emulated by a chip simulator, that provides a logging of the instruction by instruction execution in the CPU as an output file, which yields various measurement information about the execution of instructions, blocks and the whole program.

4.1.3.2 Extraction of the Execution Time

This task serves as the adapter between the output of the chip simulator and the computation process, by extracting the two important measurement information out of the output file. First of all it provides the measurements times of each instruction executed in the CPU. Second it delivers the execution time of the whole program execution. Both information are passed to the last stage, the computation of the WCET.

4.1.4 Computation of the WCET

In the last stage of the WCET analysis the previous created files and measurement results are used in order to estimate a theoretical upper bound for the worst execution time. In this approach, the foundation of this computation is a conventional static analysis, as described in Section 2.3. But in addition, the measurement results are embedded in this static analysis process.

4.1.4.1 Representation of the Program Control Flow in a Graph

In this approach the transformation of the program flow into the CFG is part of the platin analysis process. The important part of this step is the embedding of the measurements into the static analysis on instruction level. Since the previous stage measures in general more than one path there is the possibility for multiple measurements for one instruction. In this analysis this conflict is solved by computing always with the worst-case execution. If one instruction is not covered my any measured trace there are two possible ways of replacing that missing information, see Section 4.2.3.1 for further description.

4.1.4.2 Determination of the Worst-Case Path

The determination of the WCET is part of the ILP solver though IPET of platin. In summary of the whole analysis approach, the overall concept is designed in this modular way not only to simplify possible modifications, but also to facilitate the application of existing tools. The selection and integration of these tools into this hybrid analysis is described in the following chapter.

4.2 Implementation

The in the Section 4.1 described design of the three main analysis parts now have to be further specified in respect of their implementation. Therefore it is the goal of this chapter, to show which tools were chosen in this hybrid analysis and how they are embedded into the hybrid analysis.

4.2.1 Input Format of the Program

At first, the tasks of the preparation phase have to be implemented. This includes the selection of the compiler for a specific architecture, the coding of the flow facts into the source code and the partitioning of the programs control flow.

4.2.1.1 The RISCV Architecture

Since the results of the preparation phase are passed to the later stages of the hybrid analysis, the right compiler has to be chosen to fit the needs of the measurement and the computation phase. Further, the decision about the CPU architecture, on that the program later gets executed on, has to be made. For CPU architecture the RISC-V ISA RV32IM [13]. Compiled was the program with an clang front end based on the LLVM toolchain [8].

4.2.1.2 Pragma Platina Flow Facts

The process of specifying the flow facts in the source code of the program is showed with the following example Listing 4.1. Since it contains an if-else statement followed by a loop, the control flow of function `f` is equivalent to the one in the CFG depicted in Section 4.1.4.2. Like described in Section 4.1.4.2, there is the need for information about feasible and infeasible paths through the program. In the case of a loop, this information can not be extracted solely by the analysis of the branch instructions, but must be stated in form of a loop bound. This loop bound is essential for the ILP solver in order to find the longest path through the program without assuming an infinite number of loop iterations.

In the example of Listing 4.1 the loop bound is defined by the statement depicted in Listing 4.2. By coding it in front of the loop head it gets assigned to it.

Therefore it limits the variable $x_j \leq 5$, with $0 < j < n$, which represents the edge from the node *loop body* back to its predecessor *condition*. It must be remarked, that finding a correct and tight loop bound is not always as easy as in this example, since in the case of a more complex loop condition, the maximum number of possible iterations is not as obvious as in line 6.

4.2.1.3 The Program Metainfo Language

As described in Section 4.1.2.3, it is the last task of the preparation phase to transform the program in a form, which can be used to extract the program flow. This is covered by the *clang* compiler front

```
1  int f(int input){
2    int output = 0;              // start
3    if(input > 0){               // condition
4      output += input;           // then
5    }else{
6      output -= input;           // else
7    }
8    for(int i=0;i<5;i++) {       // condition
9      output += output;          // loop body
10   }
11   return output;               // end
12 }
```

Listing 4.1 – The initial code of the exemplary program did not contain any flow facts.

```
1 #pragma loopbound min 0 max 5
2 for(int i=0;i<5;i++){ /*...*/ }
```

Listing 4.2 – In order to assign a bound to the loop, a certain statement has to be defined in front of the loop.

end, which produces a program metainfo language (PML) file. This file contains all the information about the basic blocks necessary for the build process of the CFG and gets therefore passed to the last stage [6].

4.2.2 Measurement of the Execution Time

Given that the PML file is created, the initial C file is complemented with the necessary flow facts and compiled, the next step in the analysis process handles the measurement execution. This section describes the chip emulator, that is used for this program execution, while focusing on the structure of the output file. Hereafter the process of extracting the execution time of each instruction is explained.

4.2.2.1 The Rocket Chip Simulator

For the simulation of the program execution the Rocket Chip Generator was used [1]. It contains the needed tools in order to generate designs of integrated circuits. Since its processor generators are written in a hardware construction language [1], this implementation design opens up the possibility for the evaluation of variances in the underlying hardware and its effects on the overall WCET estimation performance. Looking back to the exemplary function in Listing 4.1, a execution of a program, calling this function, would result in a step by step execution of every instruction. The ouput file of the rocket chip therefore contains the logging information of the CPU state for every cycle while the execution is simulated. For further explanation of the structure of the simulation the loop body of Listing 4.1 is consulted. On object code level the loop body block contains the instructions from Listing 4.3.

It loads the value of the *output* variable into register *a0*, adds it up with itself onto *a0*, which results in *output += output;* on C code level, and finally stores it. The last instruction of the block is the jump instruction of the second condition block, which was also the predecessor block of the loop body. Executed on the rocket chip the output file therefore contains the sequence depicted in Listing 4.4.

Each line stands for one CPU cycle, which is the first number after CPU core. In addition, in each cycle the executed instruction is given in from of its object code and position in the RAM.Further, each line shows the position of the program counter defined by the value of pc. The number left of

```
1 0x800010c0:      ff042503        lw      a0,-16(s0)
2 0x800010c4:      00a50533        add     a0,a0,a0
3 0x800010c8:      fea42823        sw      a0,-16(s0)
4 0x800010cc:      0040006f        j       800010d0 <f+0x88>
```

Listing 4.3 – Instrucions of the loop body from Listing 4.1 on object code level.

```
1 281545 [1] pc=[800010c0] ... inst=[ff042503] lw      a0, -16(s0)
2 281546 [0] pc=[800010c0] ... inst=[ff042503] lw      a0, -16(s0)
3 281547 [1] pc=[800010c4] ... inst=[00a50533] add     a0, a0, a0
4 281548 [1] pc=[800010c8] ... inst=[fea42823] sw      a0, -16(s0)
5 281549 [1] pc=[800010cc] ... inst=[0040006f] j       pc + 0x4
```

Listing 4.4 – The output of the Rocket Chip Simulator, depicting the loop body from Listing 4.3.

the program counter is of particular importance. The `[1]` at cycle `281548` for example signifies a valid instruction in the writeback stage, which is in this case the store word command at `0xfea42823`. A `[0]`, like as in cycle `281546` shows, that there is no valid instruction on the writeback stage. But the most important thing that can be seen in this example of the rocket chip output file is, that a instruction execution can last longer than one cycle in the writeback stage, like the *load word* operation at `0xff042503`. Looking at the whole log one can see that even the execution time of the same instruction can vary in the progress of the program because of cache and pipeline effects. The solution for the definition of a instructions execution time by analyzing this output file is content of the following chapter.

4.2.2.2 Parsing the Instruction Executions

Understanding the process of parsing the execution times is important in order to understand the output of this stage of the WCET analysis. The measurement was conducted on statement level, meaning that for every instruction its execution time was extracted. In addition the parsing process measures the whole program execution time for later evaluation of the hybrid analysis with a conventional measurement based WCET analysis. In this approach the execution time of one instruction was defined by the time between its first valid appearance in the writeback stage of the pipeline until the next valid instruction is in the writeback stage. This implementation approach has advantages and disadvantages. On the one hand it is a simple solution for measuring the execution time of blocks and the program as a whole, because the execution times of sum of the single instructions added up equals the execution time of the entire program respectively block. On the other hand it fails at defining the execution time of the instruction itself, as the example from Listing 4.5.

In this segment a value of a specific data element gets incremented by one in cycle 434760. For that, the memory position of this data element is calculated in the previous instructions. As it can be seen, the parser assigns the value 28 to the addi instruction of 0xe2050513, although a addi instruction is rather simple and the delay in the program execution is caused more likely by the successing add instruction at 00a58533 due to data hazard because of dependency to the value of register a0.

4.2.3 Computation of the Upper Bound

The last stage of this analysis approach is the estimation of the WCET by computing an upper bound based on the results of the previous stages, namely the PML-file, the binary file and the measurement results of the parser of the rocket chip's output file. This sections describes therefore the functionality

```
1  434728 [1] pc=[800013b8] ... inst=[00005537] lui     a0, 0x5
2  434729 [1] pc=[800013bc] ... inst=[e2050513] addi    a0, a0, -480
3  434730 [0] pc=[800013bc] ... inst=[e2050513] addi    a0, a0, -480
4  434731 [0] pc=[800013bc] ... inst=[e2050513] addi    a0, a0, -480
5  ...
6  434754 [0] pc=[800013bc] ... inst=[e2050513] addi    a0, a0, -480
7  434755 [0] pc=[800013bc] ... inst=[e2050513] addi    a0, a0, -480
8  434756 [0] pc=[800013bc] ... inst=[e2050513] addi    a0, a0, -480
9  434757 [1] pc=[800013c0] ... inst=[00a58533] add     a0, a1, a0
10 434758 [1] pc=[800013c4] ... inst=[00052603] lw      a2, 0(a0)
11 434759 [0] pc=[800013c4] ... inst=[00052603] lw      a2, 0(a0)
12 434760 [1] pc=[800013c8] ... inst=[00160613] addi    a2, a2, 1
13 434761 [1] pc=[800013cc] ... inst=[00c52023] sw      a2, 0(a0)
```

Listing 4.5 – The execution time of a single instruction can not be extracted from the rocket chip output. It only shows how long the instruction remains in the writeback stage.

and usage of the chosen tool and the embedding of the instruction execution time in its existing static analysis.

4.2.3.1 The Platin Toolkit

The applied WCET tool in this approach is the platin toolset. It contains tools for the estimation of the WCET of a program based on its PML file, which had been generated in the first stage in this process. Thereby it follows the common concept of a static analysis by generating a CFG as a formal representation of the program and modeling the underlying hardware. It covers therefore the two necessary tasks stated in Section 4.1.4. The disadvantage of platin lies in the challenge of hardware modelation. Since it has no feature of estimating cache behavior of a RSIC-V architecture implemented, it therefore always calculates with the pessimistic case of a cache miss. In the program of Listing 4.1 this could have an especially negative impact in the loop at the bottom part of the program. Because due to memory locality one would expect different execution times for different iterations. The effect of platin's missing persistence information with regard to loops can be seen in the CFG of Figure 4.2.

As one can see, the edge from the loop body back to the incrementation block and the back to the loop condition block affects the ILP by the factor 5. In an static analysis with persistence information there would be two edges between the loop body and the loop condition respectively incrementation block. One edge representing the block cost of the loop body while entering the loop for the first time and one edge with potentially lower costs for all the following iterations.

4.2.3.2 Embedding of the Measurements

In order to compensate for the insufficient hardware modellation of platin the previously gathered measurements could help out at estimating a correct yet tight upper bound. Since it can be very expensive to reach 100% statement coverage in the measurement stage of the analysis, a strategy for the not measured instruction has to be defined. Per default platin uses in that case the result of its static analysis, but by setting the option *–if-not-measured-zero*, it assigns to the uncovered

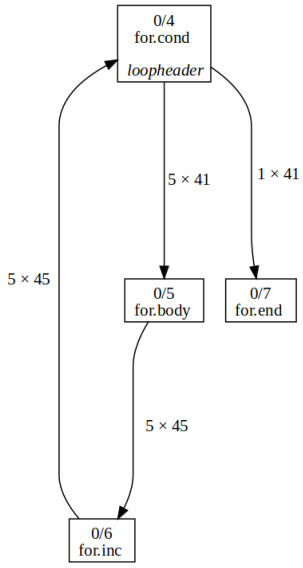

Figure 4.2 – the bottom subpart of the visualized ILP by platin (static analysis)

instruction the value zero. The final results of analyzing the exemplary program of this section, shown at Listing 4.1, can be seen Listing 4.6.

In that snippet of the final log file two blocks are depicted. The *else block*, which was not covered by the measurement and the *condition block* of the *for loop*, whose instruction were executed five times for every iteration and an additional last time at the termination of the loop. Below that one can see the entire execution time measured at the second stage and the statement coverage. The last important number is the WCET estimated by the hybrid analysis, *371 cycles*.

4.3 Summary

One strength of the analysis approach described in this chapter is its modular construction. While some other hybrid analysis methods use online aggregation of measurement data, see Chapter 3, this analysis allows the evaluation of interim results after its second stage.

```
1  ...
2  f 0/2/0 if.else :
3  0x 80001090 LW       NOT MEASURED 17
4  0x 80001094 LW       NOT MEASURED 17
5  0x 80001098 SUB      NOT MEASURED 6
6  0x 8000109c SW       NOT MEASURED 13
7  0x 800010a0 JAL      NOT MEASURED 9
8  ...
9  f 0/4/0 for.cond :
10 0x 800010b0 LW       using M 1            1 1 1 1 1 1
11 0x 800010b4 ADDI     using M 1            1 1 1 1 1 1
12 0x 800010b8 BLT      using M 4            1 1 1 1 1 4
13 0x 800010bc JAL      using M 2            2 1 1 1 1
14 ...
15 PATH 30 : 336
16 STATEMENT_COVERAGE : 0.925 ...
17 ...
18 ---
19 - analysis-entry: main
20   source: platin
21   cycles: 371
22   cache-max-cycles: 0
```

Listing 4.6 – Besides the result of the hybrid analysis the output contains information about the execution time for each block, the execution time for each measured path and the statement coverage of the previous measurement phase.

5 EVALUATION

Looking at the approach and implementation of this hybrid analysis one could expect that its results differ from these of a conventional static analysis respectively pure measurement based analysis. The main weakness of the static analysis approach of platin is the missing hardware modeling with regards to cache effects for the RISC-V architecture. The main weakness of a measurement based analysis is the uncertainty of its upper bounds. Therefore this chapter evaluates whether this hybrid approach is now able to make up for these weaknesses.

5.1 Qualitative Analysis

The first task of this chapter is the qualitative evaluation of the analysis. By doing so, one can observe the fundamental properties of the hybrid analysis and probably some critical effects that might decrease the analysis performance.

5.1.1 Setup

Before going into further details of the evaluation results, this section illustrates the overall setup of the test examples. The underlying goal is to define micro-benchmarks that are designed to trigger specific effects. In the following process, four analysis methods are applied to these benchmarks in order to compare them with regards to their estimation performance. The following section refers to the platin analysis as *static analysis*. The two hybrid analyses are called *static-based hybrid analysis*, meaning the execution of the previously defined hybrid analysis with usage of the static calculated costs for instructions not covered by the measurement and *zero-based hybrid analysis* meaning the execution of the previously defined hybrid analysis without usage of the static calculated costs for unmeasured instructions instead using zero as cost. The *measurement-based analysis* is defined as the longest execution path of the whole program measured in the second stage. The setup of this evaluation follows the design approach in terms of Section 4.1.

5.1.2 Fundamental Properties of the two Hybrid Analysis Strategies

The first evaluation compares the two hybrid strategies. The goal is to show their behavior in terms of over- and underestimation in relation to path and statement coverage. In addition, a first comparison between the static-based hybrid analysis approach and the conventional static analysis can be made.

5.1.2.1 Micro-Benchmark

In order to do that, a benchmark was designed which executes different paths of the program in dependence of its input. These paths all cover different amounts of statements in this program. Cache effects were avoided. This means that the benchmark was designed in such way that it triggers the same cache hits respectively misses for each executed path. In order to provide that, the program consists of a series of if-else statements, each performing an addition in its then-block and the whole program works on one datum, that is incremented in these passed then-blocks. So as a consequence, no measured path execution time should differ from the effects of a cache miss at the instruction's execution time. A snippet of this micro-benchmark is depicted in Listing 5.1.

The paths that were measured in the second stage all differ in respect of the number of then-blocks they pass. By that a variation in statement coverage can be attained. An illustration of that approach can be seen in Figure 5.1. While only the red path executes all blocks, each of the other paths evaluate one more if-condition as false, leaving parts of the code uncovered. Concretely, nine if-else statements were implemented, and therefore ten different paths were measured.

5.1.2.2 Results

The results of the analysis can be seen in Figure 5.2. Both figures show the comparison between one hybrid analysis and the WOET. The WOET is in this case the red path in Figure 5.1, with 292 CPU cycles. Since there is no execution path which was not measured in the second stage this WOET is the WCET. The left figure shows that the zero-based hybrid analysis computes in general too optimistic upper bounds. With increasing statement coverage and therefore increasing knowledge about the program, the overall estimation gets more and more accurate. The right figure shows, that the static-based hybrid analysis computes in general too pessimistic upper bounds. Like the zero-based analysis, its error between estimated and actual WCET decreases with increasing statement coverage. In addition, the right figure shows the outperforming of the hybrid analysis compared to a pure static analysis, since all estimations of the hybrid analysis are less pessimistic.

```
1  int f(int input) {
2    int output = 0;        // start
3
4    if(input > 1){         // condition
5      output += 1;         // then
6    }
7
8    if(input > 2){         // condition
9      output += 1;         // then
10   }
11
12   // ...
13
14   return output;         // end
15 }
```

Listing 5.1 – The micro-benchmark designed for the fundamental strategy analysis contains a series of if-else statements, each executing an addition. The goal is, to observe the fundamental properties of the analysis strategies in terms of over- and underestimation.

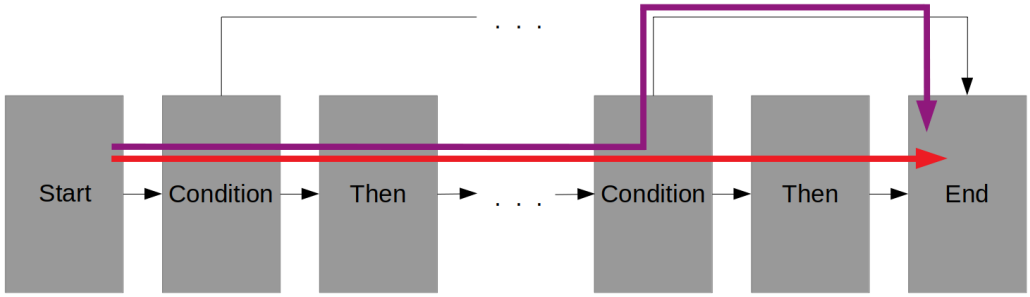

Figure 5.1 – The paths that were measured for the evaluation of the fundamental analysis properties covered all possible ways through the program.

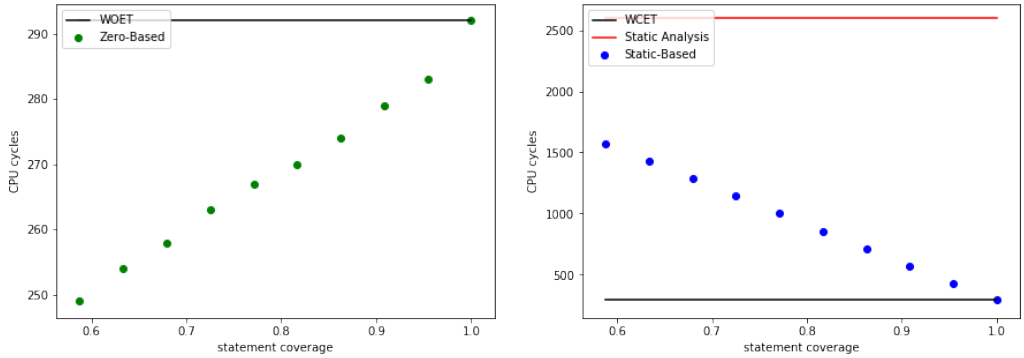

Figure 5.2 – The results of the strategy analysis show on the left side the underestimation of execution times in the zero-based hybrid analysis and on the right side the pessimistic overestimation execution times in the static-based hybrid analysis.

5.1.2.3 Explanation

In order to explain the results of the hybrid analysis, it is useful to examine how each strategy estimated the execution time of one of these then-blocks because the differences in the coverage of these blocks cause afterwards the differences in the overall estimation. The execution times calculated by the static analysis can be seen in Listing 5.2.

The static analysis does not know which of these data is cached or not. Therefore, it assumes a cache miss for each accessed data. But since each block operates on the same data the actual

```
1 0x 8000109c LW      NOT MEASURED 55
2 0x 800010a0 ADDI    NOT MEASURED 17
3 0x 800010a4 SW      NOT MEASURED 35
4 0x 800010a8 JAL     NOT MEASURED 20
```

Listing 5.2 – In the static analysis the longest possible executions times of the then-block are calculated, not measured.

execution time is due to continuous cache hits way shorter. The measured execution times are depicted in Listing 5.3

This knowledge about the cache behavior is the reason for the outperforming of the static-based hybrid analysis. Explaining the underestimation of the zero-based hybrid estimation is trivial. For every instruction that is not covered by the measurement the analysis calculates with the zero costs, making the overall estimation generally too optimistic. In conclusion, this evaluation shows the expected fundamental properties of the two analysis strategies, namely a general overestimation in the results of the static-based hybrid analysis and a general underestimation in the results of the zero-based hybrid analysis. In addition, it shows a general outperforming of the static-based hybrid analysis in comparison to the pure static analysis. One major mistake would be however to generalize the linear decreasing estimation error with increasing statement coverage that could be observed in both evaluations. This is only due to the inherent structure of the used micro-benchmark, as the following section shows.

5.1.3 The Analysis's Ability of Measuring Cache Behavior

The goal of the second experiment is to measure the effect of caching on the analysis strategies. In addition, this chapter evaluates the concept of multiple path combinations in the computation phase, which is described in Section 4.1.4.1.

5.1.3.1 Micro-Benchmark

Looking back to the benchmark of the previous evaluation, it was designed to evaluate the fundamental properties of the analysis estimation performance by deliberately eliminating different cache behavior for different paths. In opposition to that, this benchmark is designed in order to trigger cache misses and cache hits in dependence of the program input. By doing so, an evaluation of the analysis estimation performance under the cache effect is possible. This micro-benchmark's CFG is depicted in Figure 5.3. It contains two consecutive if-else statements. In both then-blocks operations on an array a are executed. In both else-blocks operations on an array b are executed. This enables four paths through the program, of which two of them trigger a cache hit and two of them a cache miss in the second if-else statement. If, for example, the path accesses array a in the first if-else block it will trigger a cache hit in the second if-else block if it operates again on array a, an operation on array b would results in a cache miss, since array b was not cached in the if-else statement before. In contrast to the previous evaluation, this time the computation phase is conducted with multiple path measurements as input. To be more exact, the hybrid analysis is evaluated on all 15 possible subsets of the four input paths.

```
1 0x 8000109c LW      using M 2          2
2 0x 800010a0 ADDI    using M 1          1
3 0x 800010a4 SW      using M 1          1
4 0x 800010a8 JAL     using M 1          1
```

Listing 5.3 – The measured execution times are due to caching way shorter then the calculated ones.

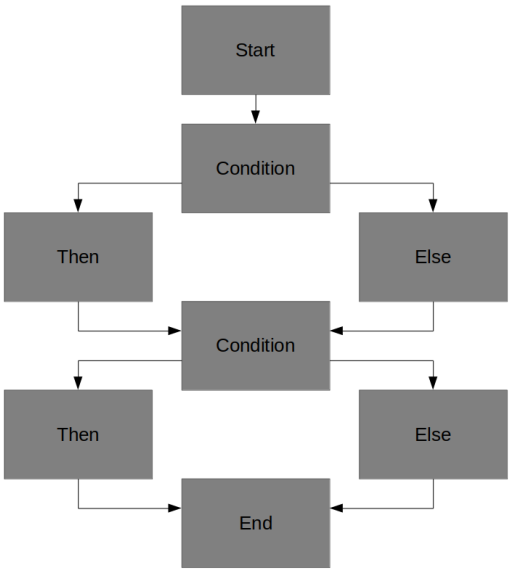

Figure 5.3 – The CFG contains two consecutive if-else statements in order to trigger cache misses and cache hits in dependence of the programs input.

5.1.3.2 Results

The results of this evaluation can be seen in Figure 5.4, depicting the estimation performance in relation to the statement coverage of the paths used in the computation phase, and in Figure 5.5, depicting the same results with regards to the number of measurement paths in the subset passed to the computation phase. Like in the previous evaluation the left sides show the performance of the zero-based hybrid analysis and the right sides refer to the static-based hybrid analysis. Looking at the results of the zero-based hybrid analysis, there is no linear relation between statement coverage and estimation error anymore. In addition, this analysis strategy does no longer produce only too optimistic results. In 9 out of 15 subsets the computation leads to an overestimation. Looking at figure Figure 5.5, it gets clear that especially those subsets with a higher number of paths lead to these overestimation in the zero-based hybrid analysis. The results of the static-based hybrid analysis again showed less pessimistic results than the conventional static analysis. But for this benchmark one subset, that consists of only two paths, leads to an underestimation. Understanding these two occasional effects, namely the unusual overestimation of the zero-based hybrid analysis and the unusual underestimation of the static-based hybrid analysis, are crucial for the correct application of the hybrid analysis. Therefore the following section describes them in more detail.

5.1.3.3 Conclusion

In order to understand why the static-based hybrid analysis underestimated the WCET in one case, one has to look at the composition of the subset, that served as input for the computation phase. Looking at the log file of that specific analysis results, retracing of the input paths is possible. In this case it was a subset containing two paths illustrated in Figure 5.6.

The special characteristic of these two paths is that they both trigger a cache hit in the second if-else statement, so the computation phase is calculating with the shorter execution times for cache

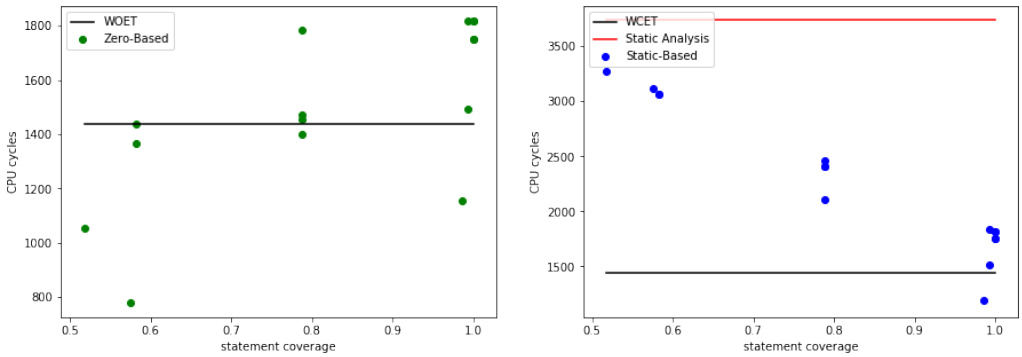

Figure 5.4 – Depicting the results of the zero-based (left) and static-based (right) hybrid analysis in relation to the statement coverage of the paths used in the computation phase.

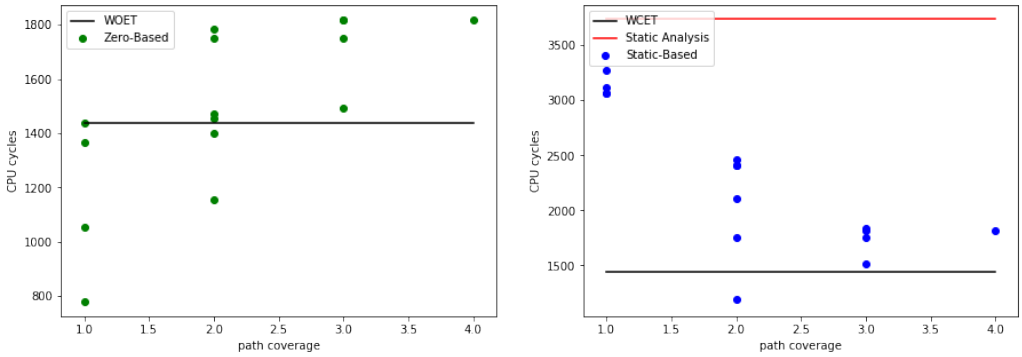

Figure 5.5 – Depicting the results of the zero-based (left) and static-based (right) hybrid analysis in relation to the path coverage of the paths used in the computation phase.

hits, missing out the information about possible cache misses. Yet, they cover all possible blocks of the benchmark, so a statement coverage of almost 100% is no longer a valid metric in order to prevent such underestimation. Like Figure 5.5 shows, an underestimation of the static-based hybrid analysis can be avoided by using subsets with a high path coverage in the computation phase. But as described in the following, this may lead to another effect making the estimation performance less reliable. Looking back to the implementation of the hybrid analysis, the computation stage calculates the execution time of a block by selecting the longest measured execution time for each of its instructions. In this way, it is possible that measured instruction executions of different measurements are combined in one block, even if this constellation of execution times in reality would be impossible. This can be seen for example in Listing 5.4 the logging of the zero-based hybrid analysis, executed with the full set of all possible paths.

This block represents the entry block of the micro-benchmark. On the right side there are all four measured paths. The middle section shows the maximum value of all of the execution times. It is these values, that add up to the execution time of the entire block. Looking at the execution paths, one can see that the third part is shorter in terms of executed instructions. In order to understand this, one has to understand how the condition in the if-else statement is implemented.

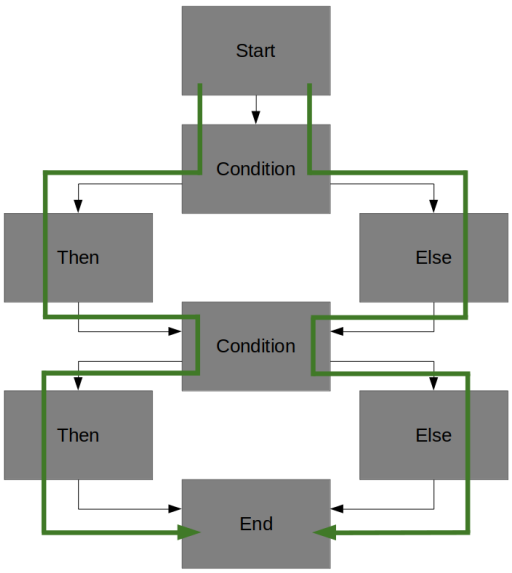

Figure 5.6 – The subset, that lead to the underestimation in the static-based hybrid analysis, contained two paths. Both of them trigger cache hits in the second if-else statement.

```
1  0x 80001048 ADDI     using M 1     1  1  1  1
2  0x 8000104c SW       using M 1     1  1  1  1
3  0x 80001050 SW       using M 1     1  1  1  1
4  0x 80001054 ADDI     using M 1     1  1  1  1
5  0x 80001058 ADDI     using M 1     1  1  1  1
6  0x 8000105c SW       using M 6     6  6  6  6
7  0x 80001060 LW       using M 1     1  1  1  1
8  0x 80001064 ADDI     using M 1     1  1  1  1
9  0x 80001068 SW       using M 1     1  1  1  1
10 0x 8000106c BEQ      using M 34    1  1  34 1
11 0x 80001070 JAL      using M 2     2  2     2
```

Listing 5.4 – In the computation phase the maximum execution times of all paths are summed up as the execution time of the whole block, leading occasionally to an overestimation.

```
1 if(control == 1 || control == 2)
```

Listing 5.5 – The condition of the if-else statement is subject to short-circuit evaluation.

The if-else condition, shown in Listing 5.5, consists of two boolean expressions combined with an or-conjunction. The shorter path of the first block is that path, that evaluates its first impression as `true` and jumps directly into the then-block due to short-circuit evaluation.

This block, depicted in Listing 5.6, represents the evaluation of the condition's second expression. Since in one path the entire condition already is evaluated as `true`, this part of code is only passed by the remaining three. In reality, the longest path through these two consecutive code blocks would be path one and four, each leading to a combined time of 79 cycles. The hybrid analysis, however, does not take into account that combining the maximum execution time of each instruction occasionally leads to impossible constellations and therefore overestimates this code section with an expected time as 114 cycles. Knowing this structural detail in the design, one can not only explain the overestimation of the zero-based hybrid analysis but also the performance decrease of the static-based hybrid analysis with the full set of paths compared to the execution with subsets only covering three paths. It seems like both effects could correlate with the number of paths passed to the computation stage. In order to gain knowledge about this correlation, the next section evaluates these two observed effects in a more quantitative approach.

5.2 Quantitative Analysis

Since the previous chapter evaluated the hybrid analysis in a qualitative way, this sections focuses at first on the evaluation of the previous observed overestimation effect quantitatively and second on the overall estimation performance on a broad set of benchmarks. At the end of this section a comparison between hybrid, static and measurement-based analysis in terms of precision is made.

5.2.1 Setup

The goal of the previous evaluation was to trigger specific hardware effects in order to see how the analysis reacts. This was done by creating micro-benchmarks. The purpose of a quantitative evaluation is to gain performance information as universal as possible. Therefore this evaluation must be carried out on the basis of randomly created benchmark, that can represent a broad set of possible real-world application of the analysis. For that the tool GENE is used [12]. GENE is suitable for this evaluation approach because it is able not only to create WCET benchmarks but to predefine its longest execution path. In addition, variables such as complexity or input length of the benchmark or specific program patterns as loops or branches can be defined.

```
1 0x 80001074 LW      using M 1        1   1    1
2 0x 80001078 ADDI    using M 1        1   1    1
3 0x 8000107c BNE     using M 60       60  30   60
4 0x 80001080 JAL     using M 2            2
```

Listing 5.6 – The second part of the condition is not executed by the path, that has already evaluated the first part of the condition as true.

5.2.2 Quantitative Evaluation of the Subset Size

The first experiment is about the evaluation of the influence of the subset size on the estimation performance. It can be expected that the results deliver a quantitative relation between subset size and estimation performance. This information is important because in the next experiment, which evaluates the overall estimation performance in relation to the conventional analysis methods, one needs to know which subset size delivers in general the optimal results in respect of underestimation and overestimation effects.

5.2.2.1 Benchmark

For this first experiment a set of 5 benchmarks were randomly created with GENE , each of them with a complexity of $cost = 1000$. The length of the input is 3 bit. Therefore there are $2^3 = 8$ different executions through each of the benchmark. Since it is initial goal of this experiment to examine the performance of each subset, the hybrid analysis was conducted for all 255 possible subset variations. The benchmarks is created with the following pattern.

- Arithmetic
- Branch
- ConstantGlobalAssignment
- Consume

Therefore the benchmarks contain branching with arithmetic instructions but no loops. They are excluded in the first quantitative experiment because their disproportional influence on the execution time could overshadow the under- and overestimation effect that are subject of this experiment.

5.2.2.2 Results

Figure 5.7 shows the experiment results for all five benchmarks. The x-axis depicts the subset size, the y-axis the estimation error. Here the *estimation error* is defined as the difference between estimated and actual WCET. Like in all previous experiments the estimation performance of the hybrid analysis executed with mono-path subsets show the fundamental characteristics of underestimation in the zero-based hybrid analysis and general overestimation in the static-based analysis again. Further, the multi-path subsets of all five benchmarks confirm the overestimation effect described in Section 5.1.3. Surprisingly this overestimation does not increase linear with a larger subset size. Instead it grows with smaller subsets and then hits a limit in every case. The results of the zero-based analysis are especially clear on that.

5.2.2.3 Conclusion

Because it seems like there is no linear correlation between subset size and overestimation it is suggested to not limit the size of the path set used in the computation phase and instead incorporate as many measurements as possible in order to minimize the risk of underestimation, see Section 5.1.3.

5.2.3 Quantitative Evaluation of the Overall Performance

The goal of this second part is to get a quantitative evaluation about the performance of the hybrid analysis compared to the two conventional analysis methods, the end-to-end measurement-based

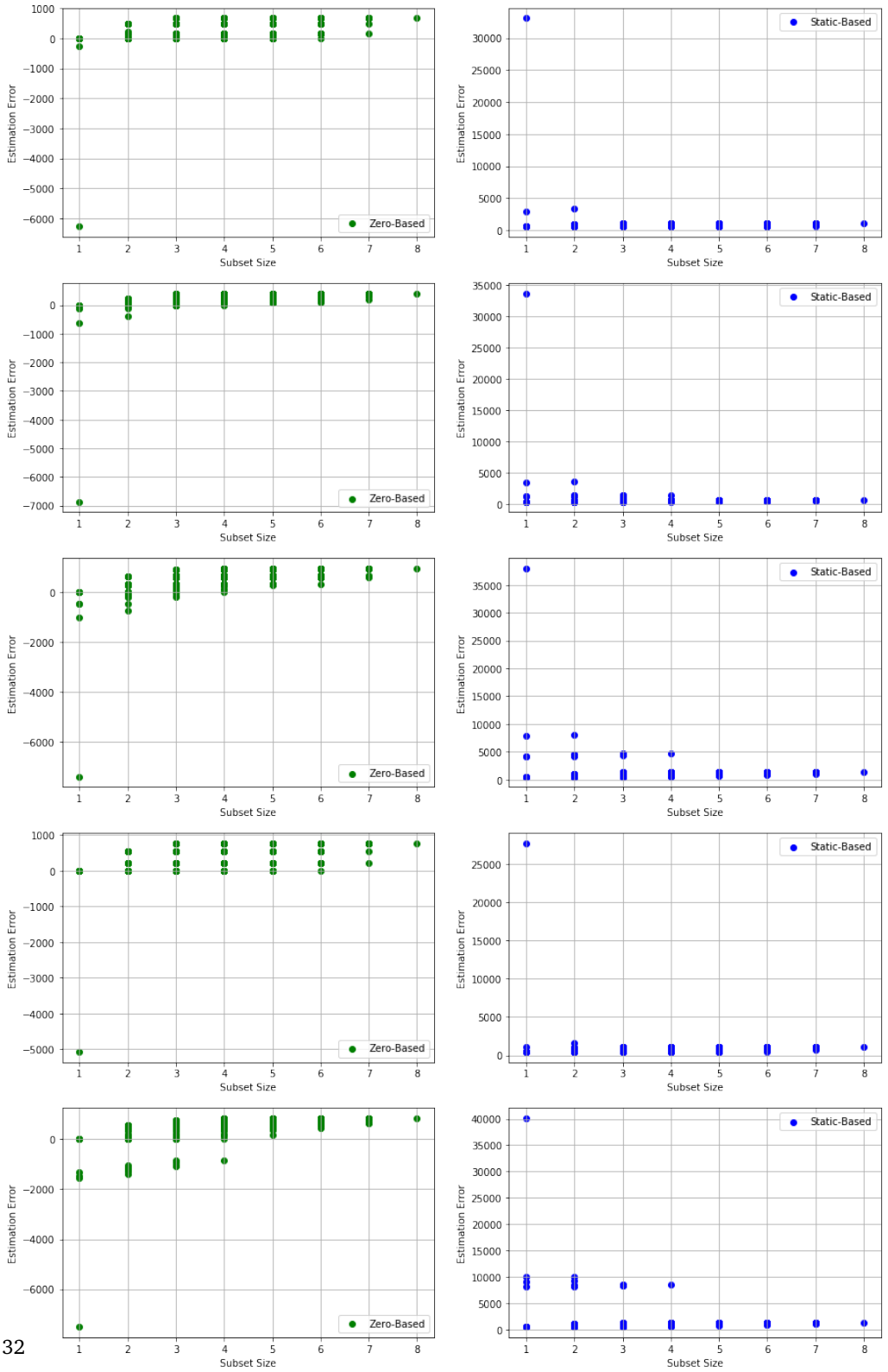

Figure 5.7 – The figure shows the analysis results for all five GENE benchmarks with the goal of evaluating the overestimation effect in a quantitative way. The x-axis shows the size of the subset and the y-axis displays the estimation error as the absolute difference between computed and actual WCET.

respectively static analysis. By doing so, this sections tries to answer the initial question of the introduction whether this hybrid approach could outperform the platin analysis on hardware for which there is no cache prediction so far implemented and at the same time provide safer results than the measurement-based analysis. Further, this evaluation gives insights on how the hybrid analysis performs for certain pattern like loops or branches and how infeasible paths influence the results.

5.2.3.1 Benchmarks

In order to answer this questions a set of four benchmarks is created with GENE , each of them with a complexity of $cost = 4000$. The length of the input is now 5 bit. Therefore there are $2^5 = 32$ different executions through each of the benchmark. This limitation in complexity, input length and number of benchmark is because of the long execution time of the hybrid analysis itself. A larger set of benchmarks certainly would increase the confidence in the performance evaluation but is due to its duration so far not possible. Therefore, the subset size has to be equally limited to five so that a simulation of the analysis's ability to compensate a lack of path coverage can be attained. Each subset was created randomly. The foundation of the benchmarks are the same pattern like in the previous section extended by the *MutualExclusivePath* pattern. In addition the four benchmarks represent the four possibilities of combining the two pattern *ConstantLoop* and *InfeasiblePath* in such way that the benchmarks of Table 5.1 are created.

NAME	LOOPS	INFEASIBLE PATHS
complex	✓	✓
only-loop	✓	✗
only-infeasible	✗	✓
simple	✗	✗

Table 5.1 – The four benchmarks represent all combinations of the pattern *ConstantLoop* and *InfeasiblePath*.

By doing so, a evaluation of the analysis performance in dealing with these kind of structures in programs is possible. And thus, the strengths and weaknesses of this thesis's approach can be identified. The given relative error is calculated as

$$estimation\_error = \frac{estimation - WCET}{WCET}$$

5.2.3.2 Results

The results of this overall performance evaluation can be seen in Figure 5.8. Each of these figures depicts the WCET estimation results with 17 conducted hybrid analyses for each of the four benchmarks described in the previous section. These are reading top-down *complex*, *only-loop*, *only-infeasible* and *simple*. Just like in the evaluation before the y-axis shows the estimation error between actual and estimated WCET time. The x-axis shows this time the statement coverage instead of the subset size since all subsets are of size five. Two things are important in this figure. At first, the zero-based analysis depicted on the left leads in a significant times to an underestimation. The static-based analysis strategy on the other hand did not show any underestimations. It performed all analysis examples within a margin of error of about 150% which is quiet small compared to the static analysis estimations depicted in Table 5.2.

NAME	STATIC ANALYSIS ERROR
complex	693 %
only-loop	442 %
only-infeasible	470 %
simple	461 %

Table 5.2 – The static analysis is less accurate than the hybrid analysis.

The WCET was measured by executing all 32 possible inputs and taking the maximum of the WOETs. In all four benchmarks the error of the static analysis is multiple times higher then the one from the static-based hybrid analysis which leads to the following conclusions.

5.2.3.3 Conclusion

The quantitative evaluation of the hybrid analysis has shown that the overestimation effect due to infeasible paths exists not only in micro-benchmarks but has a real influence in randomly created benchmarks. Yet, the static-based analysis has proven to produce tighter upper bounds than the conventional static analysis. While on the one hand the estimation error of the zero-based analysis is smaller than the one of its static-based counterpart its results lack in reliability making the zero-based approach inapplicable. The downside in this evaluation is its small number of benchmarks which limits the possibility of making further generalizations about the hybrid analysis's estimation performance with the specific pattern inside the benchmarks.

5.3 Summary of the Evaluation Results

In summary, the evaluation section has shown the benefit of the static interpolation of the static-based hybrid analysis. Its estimated upper bounds are more precise than the one of the static analysis and at the same time safer than a pure end-to-end measurement. On the other hand the qualitative evaluation with micro-benchmarks has shown the steady risk of underestimations despite high statement coverage, even for the static-based strategy.

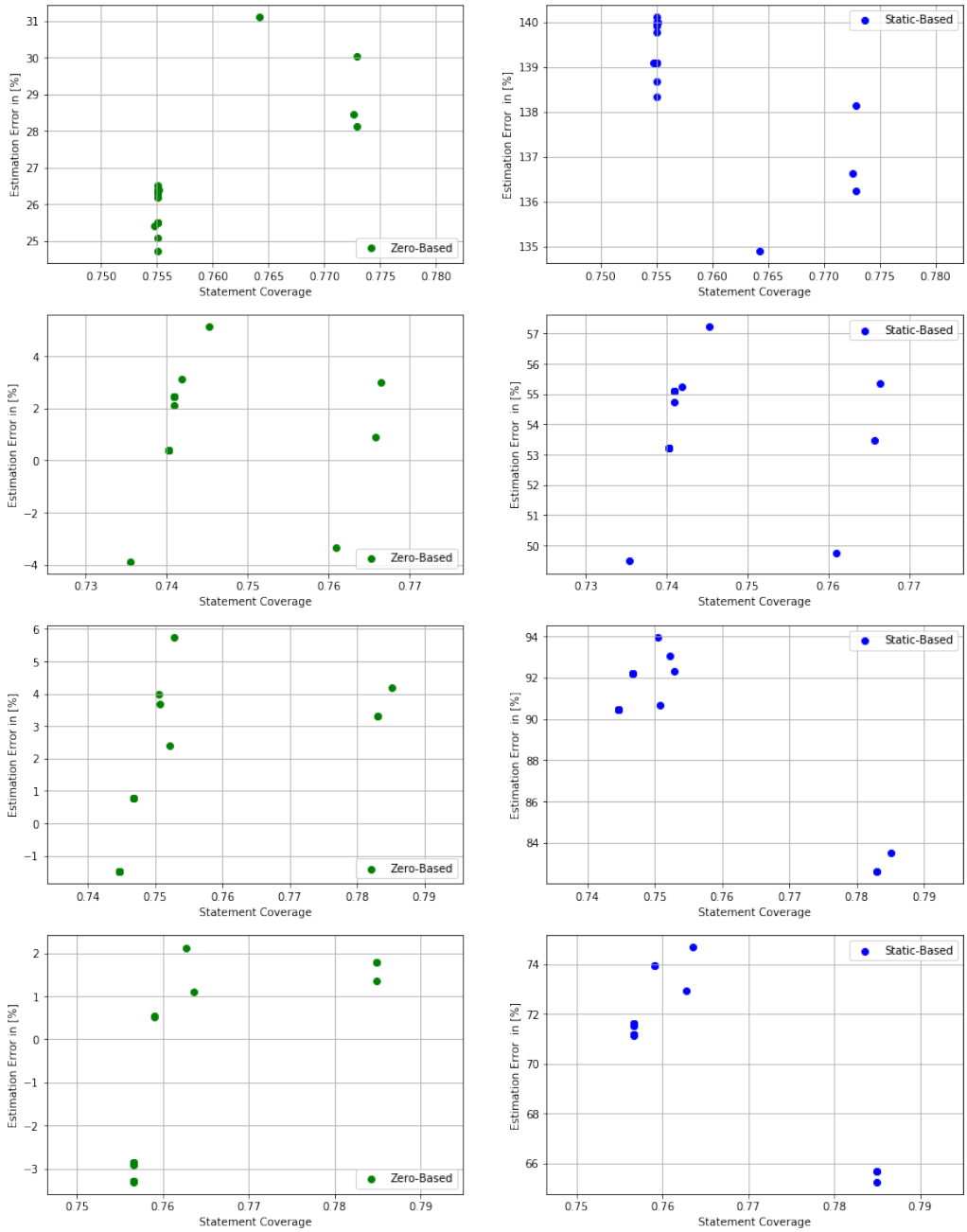

Figure 5.8 – The figure shows the overall estimation performance of the analysis for all four GENE benchmarks in the top-down order of *complex*, *only-loop*, *only-infeasible* and *simple*. The x-axis shows the statement coverage and the y-axis displays the estimation error as the absolute difference between computed and actual WCET.

6 CONCLUSION

The evaluation has shown that the hybrid approach presented in this thesis is a convenient compensation for undocumented or unpredictable processors for which a static analysis cannot be conducted. At least, the static-based approach has shown to be applicable. The zero-based analysis strategy could not provide the safe upper bounds one would expect from a WCET analysis. In conclusion, a hybrid analysis either has to be conducted with full statement coverage or the missing execution time has to be replaced by the results of a static analysis. However, this static analysis needs of course a hardware model, whose absence was the original motivation for the hybrid analysis. Therefore, this static-based hybrid analysis approach could be used as a supplement for static analysis whose hardware model is incomplete, for example, because of missing modeling of cache behavior.

The measurement aggregation on instruction level has not shown any positive influence on the estimation performance. If anything, it opened the possibility for overestimation due to composition of different execution times that cannot occur in reality. Another important observation of this thesis's evaluation is the fact that a full statement coverage does not prevent underestimations especially if the influence of cache effects on the execution time is very strong.

The main weakness of both the zero-based and static-based analysis is its long execution time for multi-path subsets which leads to the restriction of traces used in the computation phase and ultimately to low statement and path coverage. As the quantitative evaluation shows the main reason for overestimations is still the pessimism of the static analysis results used in the uncovered code. Future work could improve the runtime behavior of the computation phase with regard to the number of input traces. By enhancing the number of input traces one could not only reduce the pessimism of the analysis but also make the analysis results more reliable.

LIST OF ACRONYMS

WCET	worst-case execution time
IPET	implicit path enumeration technique
WOET	worst-observed execution time
CFG	control-flow graph
ILP	integer linear progam
PML	program metainfo language
FPGA	field-programmable gate array

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[1] Krste Asanovic et al. “The rocket chip generator.” In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[2] Sanjeev Baskiyar and Natarajan Meghanathan. “A Survey of Contemporary Real-time Operating Systems.” In: *Informatica (03505596)* 29.2 (2005).

[3] Adam Betts, Nicholas Merriam, and Guillem Bernat. “Hybrid measurement-based WCET analysis at the source level using object-level traces.” In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[4] Boris Dreyer et al. “Continuous non-intrusive hybrid WCET estimation using waypoint graphs.” In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[5] Boris Dreyer et al. “Precise continuous non-intrusive measurement-based execution time estimation.” In: *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[6] Stefan Hepp et al. “The platin Tool Kit - The T-CREST Approach for Compiler and WCET Integration.” In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS) 2015*. 2015. URL: http://publik.tuwien.ac.at/files/PubDat_246928.pdf.

[7] Daniel Kästner et al. “TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis.” In: *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*. Ed. by Sebastian Altmeyer. Vol. 72. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 1:1–1:11. ISBN: 978-3-95977-118-4. DOI: 10.4230/OASIcs.WCET.2019.1. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10766.

[8] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation.” In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[9] Yau-Tsun Steven Li and Sharad Malik. “Performance analysis of embedded software using implicit path enumeration.” In: *ACM SIGPLAN Notices*. Vol. 30. 11. ACM. 1995, pp. 88–98.

[10] Yannick Moy et al. “Testing or Formal Verification: DO-178C Alternatives and Industrial Experience.” In: *IEEE Software* 30.3 (2013), pp. 50–57. ISSN: 1937-4194. DOI: 10.1109/MS.2013.43.

[11] Serdar Tasiran and Kurt Keutzer. “Coverage metrics for functional validation of hardware designs.” In: *IEEE Design Test of Computers* 18.4 (2001), pp. 36–45. DOI: 10.1109/54.936247.

[12] Peter Wägemann et al. “Benchmark generation for timing analysis.” In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2017, pp. 319–330.

[13] Andrew Waterman et al. “The risc-v instruction set manual, volume i: Base user-level isa.” In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[14] Ingomar Wenzel et al. “Measurement-based worst-case execution time analysis.” In: *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*. IEEE. 2005, pp. 7–10.

[15] Reinhard Wilhelm et al. “The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools.” In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: http://doi.acm.org/10.1145/1347375.1347389.