To appear in: Proceedings of the 23rd Real-Time Embedded Technology and Applications Symposium (RTAS 2017)
Pittsburgh, USA, 18–21 April 2017

Benchmark Generation for Timing Analysis

Peter Wägemann, Tobias Distler, Christian Eichler, and Wolfgang Schröder-Preikschat
Friedrich-Alexander University Erlangen-Nürnberg (FAU)

Abstract—**Being able to comprehensively evaluate the individual strengths and weaknesses of worst-case execution time (WCET) analysis tools through benchmarking is essential for improving their accuracy. Unfortunately, a lack of knowledge about the detailed characteristics, actual complexities, and internal structures of existing benchmarks often prevents fine-grained assessments, and sometimes even results in misleading conclusions. In this paper we present GENE, a tool that addresses these problems by automatically generating WCET benchmarks with known properties and predefined complexities. Due to the WCETs of benchmarks created by GENE being available, this approach for example makes it possible to precisely determine the accuracy of a WCET analyzer. In addition, the fact that GENE controls the program patterns of a benchmark enables fine-grained evaluations of the particular abilities and deficiencies of different WCET analyzers, as we demonstrate for aiT and PLATIN using multiple hardware platforms.**

I. Introduction

Knowledge about the timing behavior of scheduled tasks in general, and their worst-case execution times (WCETs) in particular, is essential for building dependable real-time systems. However, precisely obtaining such information is only straightforward under certain circumstances, for example, when a task program has no input, therefore always follows the same program path, and thus has a constant execution time when starting with identical hardware state. In contrast, if the execution time of a program varies between runs because of an input-dependent control flow, determining the WCET becomes significantly more difficult. The naive approach would be to compute the maximum of the individual execution times associated with all possible inputs, which is usually not feasible due to the huge amount of input values: For an input of n bits, the program had to be executed 2^n times.

To circumvent this problem, existing WCET tools [1], [2], [3], [4], [5], [6] follow a different approach: they derive the WCET from the program's structure. That is, utilizing knowledge about the execution times of instructions on a given hardware platform, WCET tools analyze critical program patterns such as conditional statements or loops in order to gain information about the control flow, and eventually report a value for the WCET. However, as a complete analysis requires an enumeration of all possible program paths, which due to their large number is in general infeasible [7], WCET analyzers opt for less extensive analyses. As a result, for non-trivial programs the WCET reported by analysis tools usually exceeds the actual WCET; in our evaluation we observed deviations of up to a mean factor of 4.01. This means that there is potential for increasing the accuracy of WCET analyzers, ideally closing the gap between reported and actual WCET.

The common practice to assess improvements in timing analysis is to evaluate analyzers with WCET benchmark suites [8], [9], [10], [11]. However, as we will discuss in detail, this approach has several major drawbacks: First, due to the fact that for many benchmark programs contained in such suites the actual WCET is unknown and cannot be easily determined, it is impossible to assess the accuracy of a WCET analyzer (i.e., the factor between reported and actual WCET) on an absolute scale. Second, a significant number of available benchmarks is not resilient against compiler optimizations and/or does not have an input-dependent control flow. As a result, despite appearing to be of complex nature at the source-code level, these benchmarks do not pose a challenge to WCET analyzers performing their analysis on optimized machine code. Third, as existing benchmarks are often monolithic compositions of different program patterns, it is difficult to evaluate the individual strengths and weaknesses of a WCET analyzer. This, for example, makes it hard to identify the program patterns for which the analysis techniques of a particular analyzer could be improved.

In this paper, we address these problems with GENE[1], a tool that not only automatically generates WCET benchmarks with known characteristics but in addition also provides information about their properties, including for example the benchmarks' WCETs. To achieve this, GENE constructs each benchmark by relying on small program building blocks whose properties are known and composing them in a way that allows the tool to ensure that a predefined input value will lead to the WCET. Using this worst-case input, GENE in a next step is then able to determine the actual WCET of the benchmark by measuring its execution time on the target hardware platform.

Besides knowing the properties of the generated programs, constructing benchmarks from scratch has the main advantage that GENE can both enforce an input-dependent control flow and also fine-tune the complexity of the created programs. Furthermore, the fact that GENE operates at a low level and relies on program patterns that already resemble optimized machine code hinders the structure and complexity of benchmarks from being vulnerable to compiler optimizations.

In our evaluation, we show how to use GENE to reveal the individual strengths and weaknesses of two state-of-the-art WCET analyzers: the commercial tool aiT [1] and the open-source tool PLATIN [2]. Furthermore, we describe how GENE allowed us to detect a previously unknown bug that caused aiT to report WCET values that are lower than the actual WCETs.

[1]The name GENE originates from the term "genie" and the idea of Generating Evaluation sets.

978-1-5090-5269-1/17/$31.00 ©2017 IEEE

In summary, this paper makes the following contributions:

- It presents GENE, a generator for WCET benchmarks with known properties and predefined complexities.
- It discusses details of the current GENE implementation.
- It uses GENE to evaluate the two state-of-the-art WCET tools aiT and PLATIN on different hardware platforms.

The remainder of the paper is structured as follows: Section II provides background on WCET benchmarking. Section III identifies common problems associated with existing benchmarks for timing analysis. Section IV and V present details on the concept and implementation of GENE, respectively. Section VI evaluates two WCET analyzers using GENE benchmarks. Section VII discusses further usage scenarios of GENE besides WCET-tool benchmarking. Section VIII presents related work, and Section IX concludes.

II. BACKGROUND

Given a specific application program and hardware platform, WCET analyzers determine an upper bound for the execution time of the program on the targeted platform. Listing 1 illustrates the structure of a program consisting of different patterns that are challenging for state-of-the-art WCET analyzers and consequently represent typical building blocks of the programs used to benchmark these tools, for example, in the context of the WCET Tool Challenge [12], [13], [14]. Examples of such patterns include input-dependent computations, different shapes of loops, assignments, function calls, as well as conditional branch statements, because these elements make it more difficult to identify the worst-case path through the program and to determine its execution time. Apart from that, timing analysis is complicated by the fact that the WCET of a program depends on the duration of the instructions invoked and consequently is hardware-platform specific.

To facilitate the evaluation of WCET tools, different benchmarks and benchmark suites for timing analysis [8], [9], [10], [11], [15] have been created, which consist of programs structurally resembling the example in Listing 1. For example, the EMSBench [15] is modeled after a real engine-management system and mostly comprises computations, assignments, and conditional branches. In general, to reflect the characteristics of many embedded real-time systems (e.g., from the automotive domain [16]), WCET benchmarks rely on inputs with predefined sizes and do not dynamically allocate memory.

Listing 1: Pseudo-code example illustrating the structure of a challenging benchmark program used for evaluating timing-analysis tools.

```
const int N = 1024; // bounded program structure
int main(int input){
  int i, j, result = 0;
  for (i = 0; i <= N; ++i)    // constant loop
    for (j = i; j <= N; ++j) // triangular loop
      result = process(...); // process input data
  if ((input > 0) && (result > 100)) // conditional
    result = 100;                    // branch
  return result;
}
```

Benchmarking of WCET analyzers is usually performed on architectures that do not exhibit timing anomalies [14]. This means that executing a program with the same input value and an identical hardware state always takes the same amount of time and therefore ensures deterministic results. This implies that a benchmark's duration does not depend on the execution history of the processor. However, note that this does not rule out the use of hardware features such as caches or pipelining.

III. PROBLEM STATEMENT

In this section, we investigate several open issues that have to be addressed in order to enable comprehensive evaluations of WCET analyzers. This discussion allows us to formulate a set of key requirements for GENE and to outline the approaches used by our tool to fulfill them.

A. Assessing the Accuracy of Timing Analysis

Determining the actual WCET of a program in general is an inherently difficult problem, which is why the results reported by WCET analyzers often represent overestimations of the actual WCET. As a consequence, there is potential to improve the accuracy of such tools, that is, to minimize the gap between the reported and the actual WCET.

The common approach for evaluating WCET analyzers is to test them with a set of programs, usually taken from one or multiple benchmark suites. Unfortunately, with the exception of a few trivial benchmarks, the input values that trigger the WCET are unknown for these programs and consequently it is not possible to obtain their actual WCETs. This means that WCET analyzers are asked to solve a problem for which the correct solution is unavailable. As a result, without having a baseline it is generally impossible to confirm whether a reported solution is in fact correct or, if this is the case, to assess how accurate it is on an absolute scale.

Even more problematic, WCET tools on occasion require additional knowledge about the characteristics of a program in order to be able to complete the analysis of a benchmark at all; typical examples of such information are bounds of the loops contained in a program. However, similar to the actual WCET, loop bounds and other flow facts (e.g., feasible paths, recursion depths) often cannot be easily extracted from existing benchmarks as this would require an enumeration of all program paths, which is usually infeasible due to the large number of possible paths [7]. In general, determining all flow facts of a benchmark manually is labor-intensive and error-prone [17], [18] and may consequently lead to false assessments of a WCET analyzer's accuracy.

Requirement: To enable precise assessments of the accuracy of WCET analyzers, GENE must not only provide benchmarks but also their respective actual WCETs and flow facts.

Approach: Instead of relying on available benchmarks with unknown characteristics, GENE generates new benchmark programs by combining small building blocks whose worst-case paths and flow facts are known. Using a metaphor: Instead of taking an already existing labyrinth and being dependent on someone finding the path through the labyrinth, GENE defines the path and then builds a labyrinth around it.

B. Fine-Tuning the Complexity of Benchmarks

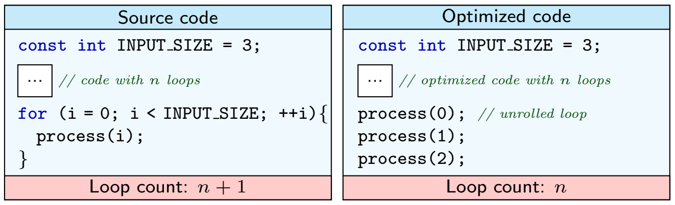
Thorough evaluations of WCET analyzers require the ability to control the complexity of benchmark programs at a fine granularity. Only this way it is possible to precisely determine the correlation between the structural characteristics of a program and the resulting duration of the analysis, for example, in order to find out at which complexity an analysis becomes infeasible or too expensive. An essential means in this context are complexity metrics [19], [20] as they allow to quantify the complexity of programs and thus to decide which benchmarks pose greater challenges to WCET analyzers than others.

Computing complexity measures such as the number of loops, function calls, or linearly independent paths for an existing benchmark program can often be automated and therefore is usually straightforward. However, the same does not apply to the problem of writing a program with specific complexity properties, mainly because complexity measures are not necessarily stable across different levels of compiler optimizations [21]. Figure 1 illustrates this issue using a simplified pseudo-code example, in which the number of loops in a benchmark is intended to be increased by adding the loop depicted on the left side of the figure to the program's source code. Inadvertently, the complexity of the benchmark with regard to loops, however, remains unaffected by this procedure, if a compiler unrolls the loop after having detected that the loop bound is fixed. As a consequence, the number of loops does not change at the optimized machine-code level at which WCET tools generally perform their analyses. This is not a problem specific to loops; similar observations of measures being altered by compiler optimizations can also be made with regard to other complexity metrics such as the cyclomatic complexity [19]. Furthermore, it is a problem actually affecting existing WCET benchmark suites [21].

Besides ensuring that the complexity of benchmark programs is actually visible to WCET analyzers, fine-tuning the complexity of benchmarks requires another crucial prerequisite to be fulfilled in order to prevent misleading conclusions: analyzed programs must have input-dependent control flows. That is, a program must comprise multiple possible paths, and function parameters and/or external variables must have an influence on which of these paths is taken during execution. If this is not the case and a benchmark program consists of only a single path, timing analysis is straightforward (i.e., running the program once and measuring the execution time), although some complexity measures might suggest otherwise.

Requirement: GENE must allow users to fine-tune the complexity of benchmarks and furthermore ensure that the generated programs have an input-dependent control flow and are not affected by compiler optimizations.

Approach: Constructing benchmarks from scratch, GENE is able to incrementally control the complexity of benchmarks and to enforce an input-dependent control flow. Due to the tool operating at a low level (i.e., the LLVM intermediate representation [22]), GENE generates benchmarks that already resemble optimized code, preventing compilers from decisively changing the structure of programs.


Source code

```
const int INPUT_SIZE = 3;
... // code with n loops
for (i = 0; i < INPUT_SIZE; ++i){
  process(i);
}
```

Loop count: $n+1$

Optimized code

```
const int INPUT_SIZE = 3;
... // optimized code with n loops
process(0); // unrolled loop
process(1);
process(2);
```

Loop count: n

Figure 1: Pseudo-code example for a complexity measure (i.e., the number of loops) not being stable across compiler optimizations.

C. Identifying Individual Strengths & Weaknesses

Task programs of real-time systems usually consist of a wide spectrum of program patterns that are challenging for WCET analyzers, ranging from different shapes of loops over value-dependent mutually-exclusive paths to infeasible paths. Due to the analyses of different patterns requiring different methods and techniques, in general it is not guaranteed that a WCET analyzer that, for example, is able to handle rectangular loops at the same time also effectively copes with triangular loops. Instead, with regard to program patterns, WCET analyzers have individual strengths and weaknesses, and clearly identifying them would greatly help researchers and developers in improving the quality of these tools.

Unfortunately, obtaining such information about particular patterns based on traditional benchmarks is cumbersome [18], if possible at all [23]. This is mainly due to the fact that existing benchmarks are often monolithic compositions of multiple different program patterns and consequently do not allow fine-grained evaluations of specific patterns. As a consequence, instead of enabling useful conclusions such as "WCET analyzer A has difficulties analyzing pattern P", evaluations with traditional benchmarks mostly provide limited insights such as "WCET analyzer A has difficulties analyzing benchmark B", which is not very informative if the benchmark consists of a variety of patterns whose effects interfere with each other.

In addition to the techniques used for analyzing program patterns, the individual quality of a WCET tool to a great extent depends on its internal hardware model describing the timing behavior of the target platform. If the model incorporates detailed knowledge about the platform, an analyzer is likely to provide results that closely resemble the behavior on the real hardware, especially for programs that do not require complicated analyses. On the other hand, comparably simple models that need to make conservative assumptions about a platform's timing behavior in general lead to inferior results.

Requirement: To provide the basis for fine-grained WCET-tool evaluations, GENE must support the systematic evaluation of challenging program patterns as well as means to assess the quality of an analyzer's hardware model.

Approach: GENE enables users to individually select the types of patterns generated benchmarks should consist of. Disallowing the use of all complicated patterns, for example, causes GENE to create programs for which the analysis quality highly depends on a WCET tool's hardware model.

IV. GeneE

In this section, we present GenE, a benchmark generator for timing analysis, and discuss details on how the tool meets the requirements identified in Section III.

A. Overview

GenE generates benchmarks by combining program patterns (e.g., different shapes of loops, assignments, conditional branches) that are characteristic for real-world applications as well as WCET benchmarks. To build GenE's pattern library, we conducted a thorough study of both literature [24] and existing benchmark suites [10], [11], [15] in order to collect realistic patterns that pose challenges to state-of-the-art WCET analyzers and therefore, for example, have appeared in application and/or benchmark programs utilized in several WCET Tool Challenges [12], [13], [14].

Figure 2 shows an overview of how GenE, based on these patterns, produces benchmarks for which the flow facts (e.g., loop bounds, value ranges, feasible paths, recursion depths) and the actual WCET for a specified target hardware platform are known. In a first step, GenE's main component, the benchmark generator, constructs the benchmark program in such a way that a predefined input value will trigger the worst-case path through the program. Using this worst-case input value, in a next step GenE determines the actual WCET by executing the benchmark on the target hardware.

Composing a benchmark in GenE is an iterative process in which the benchmark generator incrementally builds the program by repeating the following steps: First, the generator randomly selects the next pattern to insert from a set of usable patterns provided by the pattern library. Second, it weaves the selected pattern into the emerging program, thereby ensuring that the designated worst-case path actually leads to the WCET on the target platform. Third, the generator updates the flow facts of the benchmark to reflect the effects of the new pattern.

GenE provides users with different opportunities to influence the structure and complexity of the generated benchmarks. Most importantly, users are able to specify a *path budget*, that is, a unit-less value representing the length of the resulting worst-case path, which also serves as termination criterion for the generation process. In addition, users may configure the types of patterns to be used allowing them to assess the individual strengths and weaknesses of a WCET tool. For convenience, GenE for this purpose offers a number of built-in *pattern suites*, which are predefined property configurations that result in benchmarks with certain characteristics.

B. Generating Benchmarks with Known Properties

In the following, we present details on how GenE ensures that the worst-case input specified by the user leads to the WCET when the generated benchmark is executed on the target platform. Furthermore, we elaborate on how GenE determines the actual WCETs of benchmarks.

1) Constructing the Worst-Case Path: GenE automatically creates benchmark programs with known properties, the most important of which being the input value corresponding to the worst-case path through the program. To construct this worst-case path, GenE's benchmark-generation algorithm relies on a unit-less path budget specified by the user, which at all times represents the length of the remaining part of the worst-case path that still needs to be integrated into the program. At the beginning of the benchmark-generation process, this budget is at a maximum. Whenever GenE weaves a new statement or pattern into the emerging benchmark, it reduces the budget to reflect the size of the new element. The generation process terminates once the path budget reaches zero.

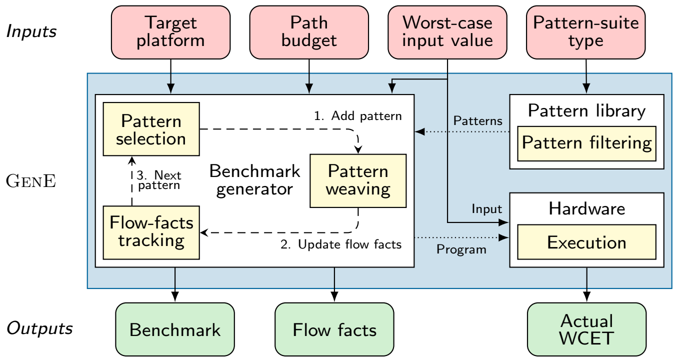
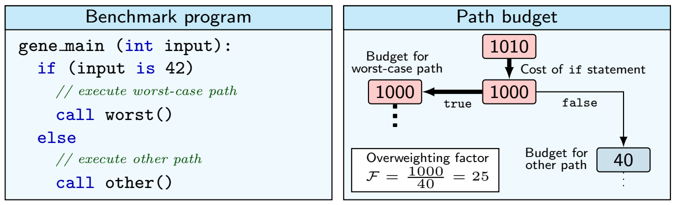


Figure 2: Overview of GenE

As shown in Figure 3, each time GenE introduces a conditional branch statement into the program, it first decides which branch W of the newly introduced branches should be part of the worst-case path and specifies the corresponding condition in such a way that the designated worst-case input (i.e., 42 in the example) results in the execution of branch W. Then, GenE assigns the remaining path budget to branch W. For all other branches, GenE creates new individual path budgets whose sizes are smaller than the remaining budget for the worst-case branch W to guarantee that the resulting paths are shorter than the worst-case path.

In order to ensure that the execution time of the constructed worst-case path is in fact higher than the execution times of all other feasible paths, GenE relies on knowledge about the timing behavior of the target hardware platform. Note that in contrast to the hardware models required by WCET analyzers, which must include absolute values of the execution times of instructions, the information on the target platform's timing behavior used by GenE can be relative [25]; that is, when constructing the worst-case path, it is for example sufficient for GenE to know that an instruction I_a takes at least x times longer than an instruction I_b, independent of the absolute execution times. In the same way, GenE is able to consider the effects of hardware features such as pipeline stalls or cache misses: By overweighting the path budget of the worst-case branch W compared with the path budgets of other branches, the benchmark generator can prevent short paths from having longer execution times than the worst-case path. For this purpose, GenE relies on a platform-dependent overweighting factor $\mathcal{F}$, which pessimistically assumes that a cache miss occurs on every instruction on the short path.


Benchmark program

```
gene_main (int input):
  if (input is 42)
    // execute worst-case path
    call worst()
  else
    // execute other path
    call other()
```

Figure 3: Pseudo-code example of how GENE controls that a specific worst-case input value (here: 42) leads to a known worst-case path. Only the branch on the worst-case path is assigned the full remaining path budget, while the other branch continues with a fraction of it.

For the Cortex-M4 hardware platform used in our evaluation, for example, GENE applies an overweighting factor of $\mathcal{F} = 25$ at the LLVM representation level. This value is a result of our analysis of the Cortex-M4 specification and the insight that, at the machine-code level, a maximum distance of 10 between all instructions is sufficient to compensate for a single instruction in the worst case. Furthermore, having compared the number of memory accesses at both representation levels, we can conclude that an overweighting factor of 25 is also large enough to account for the additional overhead introduced by potential spill code that may be generated when lowering the LLVM intermediate representation to machine code.

2) Determining the Actual WCET: Having generated the benchmark program, GENE in a next step determines the actual WCET of the benchmark by executing it on the target platform with the predefined worst-case input value. The rationale behind this approach is the insight that the most accurate execution-time model is the processor itself [26], because it inherently includes all complex hardware features such as pipelining or caching. Furthermore, measuring the actual WCET based on a worst-case execution of the program frees GENE from the need of having to rely on the documentation of a processor's timing behavior, which can be inaccurate [27]. That said, if a cycle-accurate instruction-set simulator is available for the target platform, GENE is also able to use it to determine the actual WCET by tracing the program with the worst-case input value.

In order to guarantee valid results, when generating a benchmark program GENE also creates a section that is executed prior to the actual workload and responsible for putting the system in a well-defined state that does not depend on the execution history of the processor. In particular, in the course of this preparation process GENE voids hardware state (e.g., by flushing caches) and initializes data structures that are relevant for the benchmark. This way, GENE ensures that the constructed worst-case path will in fact be executed and that the determined actual WCET is therefore valid.

C. Composing Complex Benchmarks

Below, we describe how GENE shapes the complexity of generated benchmark programs and discuss how this complexity is protected against compiler optimizations.

1) Combining Patterns: GENE's pattern library comprises a wide spectrum of different challenging program patterns including (but not limited to) assignments, computations on registers, conditional branches, as well as different shapes of loops. By combining these patterns, GENE is able to create benchmarks of different varieties and complexities. As an additional benefit, the approach to use representative structural patterns extracted from existing code allows us to circumvent the problem of some existing industrial benchmarks being affected by licensing restrictions [11]. Listing 2 shows an example of such a pattern, representing a typical method of how real-world industrial applications perform initialization work. Due to the execution time of the function `func()` depending on the value of a global variable, this pattern poses a challenge to state-of-the-art WCET analyzers.

To generate a complex benchmark, GENE weaves multiple program patterns together. For this purpose, each pattern comprises one or more insertion points, that is, code locations at which additional patterns can be added; the pattern in Listing 2, for example, possesses two insertion points: one inside and one after the body of the conditional statement. During the weaving process, GENE relies on a pseudo-random number generator to recursively determine which pattern to apply next. This way, for the same user inputs and number-generator seed, the tool will always create the same benchmark.

Having woven a new pattern into the emerging program, GENE updates the flow facts of the benchmark to reflect the extended control flow. This procedure is facilitated by the fact that each pattern in the pattern library not only contains information on its structure and insertion points but also maintains its own flow facts in a parametric way.

As discussed in Section IV-B1, GENE controls the length of program paths by decreasing the associated path budget on each pattern insertion in order to account for the size of the new pattern. For many simple patterns with constant sizes (e.g., assignments) this is straightforward. On the other hand, there are patterns such as the one in Listing 2, whose effective size in part depends on the context in which the pattern is used inside a program. In this particular case, the first call of function `func()`, for example, takes more time than all subsequent calls due to executing the body of the conditional statement. To address this issue, when constructing a benchmark GENE not only keeps track of the program's flow facts but also maintains additional context information, which includes the values of local and global variables as well as function parameters. As a consequence, GENE is able to correctly attribute the lengths of program paths even in the presence of context-sensitive patterns.

Listing 2: Code blocks that are executed only once challenge the capability of WCET tools to track the values of variables.

```
static bool initialized = false;
void func(){
  if (!initialized){
    initialize_hardware(); // insert patterns
    initialized = true;
  }
  ... // insertion point for further patterns
}
```

2) Making Benchmarks Resilient Against Optimizations: Without applying additional measures, compiler optimizations can have a significant effect on the complexity of a benchmark program (see Section III-B). To address this problem, GENE generates benchmarks at a low-level, but target-independent, representation of the program code (i.e., the LLVM intermediate representation [22]) and relies on program patterns that already resemble optimized code, thereby hindering program structures from being decisively changed by compilers. Furthermore, this approach has the additional benefit of facilitating the tracking of flow facts, which is more difficult at a higher level of abstraction such as C code [28].

Besides operating at a low level, GENE applies further measures to maintain the complexity of a benchmark across compiler optimizations in general and dead-code eliminations in particular: First, GENE uses all variables it introduces into a program. Second, besides returning the result of a computation at the end of a function, GENE also writes the value to global memory to protect the function from being removed by the compiler. Note that this not only strengthens the resilience of a benchmark against optimizations, but also imitates the behavior of applications in real systems, which communicate with external devices by writing to memory-mapped registers.

D. Selecting Patterns to Reveal Strengths & Weaknesses

GENE allows users to select the patterns that are included in the construction of a benchmark program and consequently offers the possibility to reveal the individual strengths and weaknesses of WCET analyzers. For convenience, GENE provides a number of built-in pattern suites, that is, predefined property configurations that lead to programs with certain characteristics. In the following, we discuss examples of such suites aimed at assessing the quality of an analyzer's hardware, program flow, and value analysis, respectively.

1) Challenge of Target-Hardware Analysis: As discussed in Section III-C, a WCET analyzer's model of the target hardware serves as basis for the overall timing analysis. To assess the quality of this model, GENE offers a pattern suite that results in the generation of benchmarks that consist of branchless sequences of code and only comprise assignments and computations. As a consequence of containing only a single path, these programs are trivial for the high-level (control-flow) part of the timing analysis, therefore shifting the focus on the hardware model. If the hardware model of a WCET analyzer is accurate, the results reported by the tool for the benchmarks generated by this pattern suite will closely match the actual WCETs determined by GENE.

2) Challenge of Value Analysis: An important property of WCET tools is the ability to perform value analysis in order to refine the path analysis. For example, when analyzing the code in Listing 2, tools that are unable to perform context-sensitive value analyses include the initialization part each time function `func()` is called. Due to maintaining context information on the data flow of variables when combining patterns (see Section IV-C1), GENE is able to selectively insert challenging patterns, such as infeasible paths with value constraints.

3) Challenge of Bounding Loops: Bounding the program flow for loops is essential for WCET analyzers. To evaluate the abilities of WCET tools in this regard, GENE offers several pattern suites, each targeting a different challenging loop pattern [24]. Besides the respective pattern, benchmarks generated with these suites only contain arithmetic operations and assignments, which are necessary for the body of the loop. Examples of challenging loop patterns include loops, for which the iteration variable is modified once within the loop's body (in addition to the loop's header). From such loops, no closed-form expression can be determined, which poses a challenge to WCET tools.

V. IMPLEMENTATION

The GENE benchmark generator is implemented on top of the LLVM compiler infrastructure [22]. The benchmarks are generated using LLVM's intermediate representation and the infrastructure to insert instructions into the control-flow graph. Using this representation has the major benefit that the programs are closely related to machine code (i.e., the program structure is similar in both representations), while still being independent of concrete target architectures.

A. Maintaining Flow Facts & Control-Flow Relations

Maintaining all information about flow facts (e.g., loop bounds, path constraints) across code representation levels is a challenging problem, especially when compiler optimizations decisively change the control flow. Neglecting such transformations could invalidate the flow facts gathered during the generation stage. The larger the gap between the representation level of generated benchmarks and machine code, the more difficult it is to precisely transform flow facts. Our approach requires flow facts on machine-code level where the WCET analysis is performed. As a consequence, the generated flow facts on LLVM representation level need to be transformed to machine code. To solve this issue, we exploit control-flow–relation graphs (CFRGs) [28], which are integrated into the PLATIN WCET-analysis toolkit [2]. CFRGs provide a formalized way to map the flow facts from intermediate representation to machine code. With this approach, GENE profits from generating target-independent benchmarks while operating on a low-level code representation.

GENE stores the flow facts from the generated benchmark using PLATIN's Program Meta Language (PML) format. Besides WCET analyses, the PLATIN toolkit is able to transform flow facts between different formats: This feature allows GENE to convert flow facts from the PML representation to aiT's flow-fact format (ais). With this setup, we are able to support both WCET analyzers, PLATIN as well as aiT.

B. Pattern Weaving

For proper modeling of side effects in repeatedly executed code, GENE uses an extended top-down weaving process as presented in the following. Listing 3 exemplifies the challenges of side effects in the generation process: The function `func()` is assumed to behave differently on every execution. Listing 2

Listing 3: Pseudo-code example that illustrates the necessity of context-sensitive flow facts due to the side effect of `func()`

```
for(i = 0; i < N; ++i) {
  func(); // func() behaves differently during
          // the iterations i = 0 and i > 0
}
```

depicts an example for such a function. This function is called within a loop with constant bound `N`. The budget consumed by this loop, however, cannot be calculated as `N` times the costs of a single execution of `func()`, as the cost varies for every execution of the function. To solve this problem, every pattern not only generates code, but also provides a *pattern descriptor*. The pattern descriptor is used to simulate the pattern's behavior, depending on the current execution context, such as the value stored in the `initialized` variable for the function `func()` shown in Listing 2. The context-sensitive budget consumed by the loop's body in Listing 3 is calculated as the summarized costs of the `N` simulations of the function.

The weaving process, including the simulation of patterns, is shown in Listing 4: A pattern is selected randomly and its code is produced. During the production of a pattern, further patterns can be generated by a recursive call to the function `generate()`. Generating a loop pattern, for instance, will result in generating LLVM IR for the loop itself, along with one or more productions of further patterns into the loop's body. Producing a pattern yields a pattern descriptor that describes the behavior of the pattern. Pattern descriptors are able to hold further pattern descriptors. For instance, the descriptor for the loop pattern contains the list of pattern descriptors produced into the loop's body. This descriptor is then used to simulate the behavior and obtain the costs of the previously generated code. The costs obtained from the simulation are deducted from the remaining budget, these steps are repeated until the whole path budget is depleted.

Listing 5 illustrates the simulation process of pattern descriptors of the program pattern presented in Listing 2 (conditional execution of initialization code). For this purpose, the pattern descriptor stores its child pattern descriptors into logic groups. The first group (`initBlock`) contains all pattern descriptors for patterns generated into the guarded initialize block (Listing 2, Line 4) of function `func()`. The second group (`bodyBlock`) contains all patterns generated into the unconditionally executed part of the function (Listing 2, Line 7). These blocks are taken into account during the simulation of the encapsulating pattern (Listing 5, Lines 5 and 9). In addition, the descriptor contains information about the pattern's context, such as the state of the `initialized` variable (Listing 2, Line 1). Depending on whether the initialization was already executed, the patterns in the initialization block are included in or excluded from the budget consumed by this particular execution of `func()` (`usedBudget`). The costs for both lists of patterns are determined by adding up the simulation results. This simulation of descriptors allows GenE to generate code with context-sensitive behavior.

Listing 4: Weaving process including pattern simulation

```
void generate(context, budget) {
  while(budget > 0) {
    pattern    = select_pattern(context, budget);
    descriptor = pattern->produce(context, budget);
    budget    -= descriptor->simulate();
  }
}
```

Listing 5: Simulation of descriptor `(InitOnceFunction)`

```
InitOnceFunction_PatternDescriptor::simulate() {          1
  int usedBudget = 0;                                     2
                                                          3
  if(theDescriptor.initialized == false) {                4
    usedBudget += simulate(theDescriptor.initBlock);      5
    theDescriptor.initialized = true;                     6
  }                                                       7
                                                          8
  usedBudget += simulate(theDescriptor.bodyBlock);        9
                                                          10
  return usedBudget;                                      11
}                                                         12
```

GenE does not join unrelated pieces of code, but it generates interconnected code that consists of different challenging program patterns. Supporting interconnecting patterns requires knowledge about the influence of a particular pattern on the global execution context. For this purpose, in addition to pattern descriptors, GenE also tracks the ranges of potential values for variables generated by each pattern. As a key benefit, the information obtained from this process allows patterns to reuse variables introduced by other patterns. For instance, the result of an arithmetic expression can be used by a loop pattern to calculate its upper iteration bound. Analyzing code resulting from this approach requires WCET analysis tools to track the potential values of variables in order to provide accurate results. Optimized code typically does not contain unused calculations or write operations to no longer used memory locations. Reusing the results of previous calculations further reduces the differences between code generated by GenE and optimized code produced by a compiler.

C. Available Patterns

GenE provides several program patterns that can be subdivided into the following three categories:

1) Loop Patterns: This category comprises different shapes of loops. The least complex loop is the loop pattern with a constant iteration bound, which is implemented by the pattern `ConstantLoop`. GenE's value tracking enables to generate loops whose iteration bound depends on a value computed on preceding execution paths (`InputDependentLoop`). The pattern `TriangularLoop` generates two loops that are connected by the fact that the iteration variable of the inner loop depends on the current iteration of the outer loop. The `DownsamplingLoop` is a loop containing a conditional modification of the iteration variable in the loop's body, hindering the determination of a closed-form expression for the iteration variable. If a WCET tool is not able to bound the flow of a loop and consequently cannot continue its analysis, GenE is able to provide the flow facts for the loop.

2) Path Patterns: GenE provides several program patterns influencing the control flow of the generated benchmark. The most basic of these patterns is the conditional branch pattern (`Branch`) for which GenE applies overweighting, that is, the mechanism used to construct the worst-case path (see Section IV-B1). The mutual exclusive path pattern (`MutualExclusivePath`) is an extension to the conditional branch: Two evaluations of contradicting conditions are used to disguise the exclusiveness of the two blocks, which is a challenge for analyzers' ability to perform value analyses. In addition, GenE exploits its value tracking abilities to generate dead code by crafting a non-trivial condition that is known to be unsatisfiable (`DeadCode`). Furthermore, GenE provides the previously discussed pattern `InitOnceFunction` that generates a function with a code block executed once, thereby mimicking hardware initializations (see Listing 2).

3) Atomic Patterns: These patterns consist of arithmetic calculations (e.g., addition, division) and bitwise operations (e.g., and, or, exclusive or). Furthermore, assignment operations to local and global variables are part of this category.

D. Pattern Suites

For convenience, GenE provides the following pattern suites: The `simple` suite only contains patterns that analyzers are expected to handle without additional loop-bound information. For this purpose, the only considered loop pattern in this suite is `ConstantLoop`. In contrast to the `simple` suite, the suite `valueanalysis` contains the most difficult patterns GenE provides to examine the analyzer's ability to conduct value analysis. Loops with non-constant bounds or without closed forms, such as the `InputDependentLoop` and the `DownsamplingLoop`, are of major importance for this suite. The suite `nobranch` is designed to assess the accuracy of the analyzer's hardware analysis without having influences of path analysis. Only patterns without branches (e.g., calculations and assignments) are part of this suite. The four suites `constantloop`, `inputdependentloop`, `downsamplingloop`, and `triangularloop` consist of the respective loop shape, along with arithmetic and load-store operations. In our experiments in Section VI-E3, we use these types of pattern suites to identify different WCET analyzers' strengths and weaknesses with regard to loop-bound analysis.

E. Sanitizing Generated Benchmarks

GenE has two mechanisms to internally validate that a generated program is correct: First, the tool interprets a benchmark's intermediate representation while monitoring the flow facts. During this interpretation, GenE executes the program with the worst-case input and validates the sequence of executed basic blocks, which has been set up during the generation process. Second, GenE simulates the benchmark with each bit of the input set to one and the others set to zero, while asserting that the paths hold valid context-sensitive value constraints. These features validate the correctness of the generated code. After a successful validation, GenE determines the actual WCET of a benchmark by measuring the execution time of the path triggered by the worst-case input. In addition, the tool executes the program with a configurable number of randomly selected inputs. If any of these traces were to lead to a duration longer than the execution time of the alleged worst-case input, GenE would report an error.

VI. Evaluation

Below, we rely on benchmark programs generated by GenE to evaluate two state-of-the-art WCET analyzers: the commercial tool aiT [1] and the open-source tool Platin [2].

A. Experimental Setups

For our experiments, we use two different setups, Cortex-M4 and Patmos, which are further described in the following.

1) Cortex-M4 Setup: Our main experimental setup comprises an Infineon XMC4500 development board with an ARM Cortex-M4 processor [29]. We used 20 of these boards in parallel to speed up our evaluations. To our knowledge, this commercial off-the-shelf board is one of few available and predictable hardware platforms, where the instruction timing, pipeline behavior, and memory-access latencies are documented at a fine-grained level [30], which is essential for performing accurate WCET analyses. Due to its modular structure, we were able to integrate these timings into Platin in order to enable a comparison with aiT. The 32-bit processor uses a 3-stage pipeline, contains 1024 KB on-chip flash memory with 4 KB instruction cache (2-way set associative, LRU cache-replacement policy), and runs at 120 MHz.

2) Patmos Setup: In order to evaluate Platin for an additional hardware platform, we rely on Pasim, a cycle-accurate simulator for the time-predictable Patmos processor [31]. In this setup, the processor is configured with 2kB of data cache, 2kB of instruction cache, and runs at a frequency of 80 MHz.

Both setups are comparable to the platforms used in the context of the WCET Tool Challenge [14].

B. Validating GenE

Prior to evaluating the accuracy of WCET tools, we first conduct an experiment to validate that the WCET value provided by GenE in fact constitutes the actual WCET of the corresponding benchmark. For this purpose, we give GenE a path budget of 20,000 and instruct the tool to generate a benchmark with a reduced input size of 20 bits (instead of the default 32 bits). The reduction of input size is necessary to enable us to determine all possible execution times of the benchmark by invoking it with all the about 1 million possible input values. Despite the input-size reduction, the experiment takes 12 hours for Patmos on a 48-core machine.

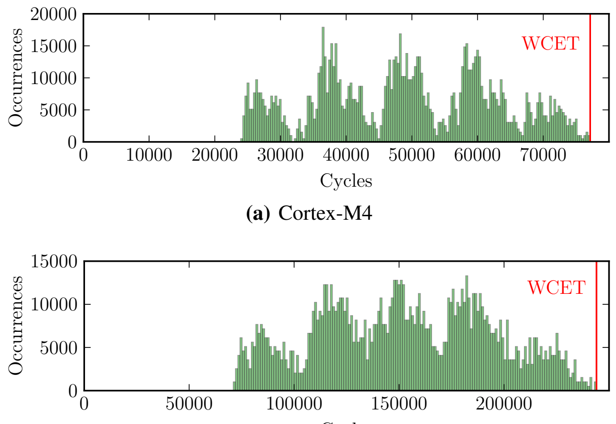
For the two evaluated platforms, Figure 4 summarizes all possible execution times of the generated benchmark as well as their occurrences. The results in both cases show a wide spectrum of execution times, indicating the existence of a multitude of different possible paths through the program. Of all input values, only 513 trigger the actual WCET in the Cortex-M4 setup (Patmos: 129), which for both setups is equal to the respective WCET value determined by GenE, thereby confirming the validity of our approach.

Since GENE relies on accurate measurements to determine the actual WCET, the underlying hardware platform needs to be predictable. In order to capture potential fluctuations in execution times caused by machine variability, we repeat the explicit enumeration of all 2^{20} possible paths 1000 times on the Cortex-M4 hardware platform. Our results show that the execution cycles are identical for the respective input values across all experiments. As a consequence, the Cortex-M4 setup behaves predictably in our experiments, making it a valid platform for benchmarking WCET analyzers.

C. Assessing the Accuracy of WCET Analyzers

In contrast to existing benchmark suites (see Section III-A), GENE not only provides benchmark programs but also their flow facts and actual WCETs. As a result, GENE offers the possibility to assess the accuracy of WCET tools on an absolute scale, using the actual WCETs as baselines against which to compare the upper bounds reported by WCET analyzers. In our next experiment, we conduct such an evaluation for aiT and PLATIN based on a benchmark with an input size of 32 bits, which therefore is more complex than the benchmark used in Section VI-B. Consequently, it is necessary to provide both WCET analyzers with all available flow facts in order for them to be able to complete their timing analyses.

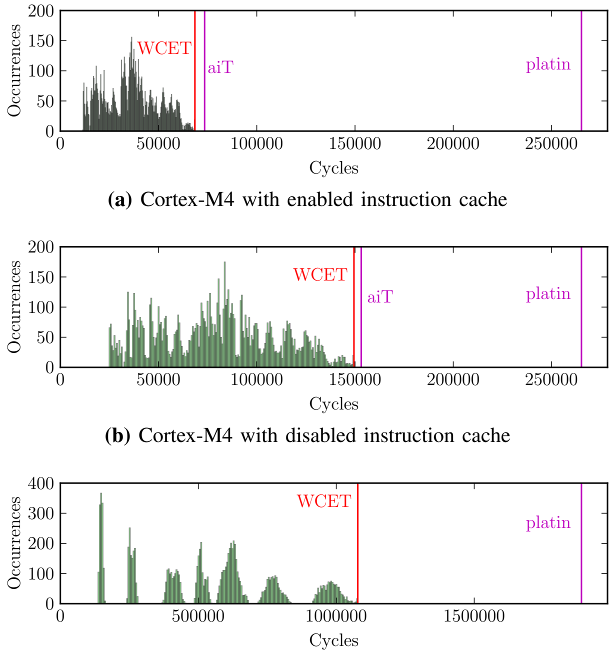
Figure 5 shows the actual and reported WCETs for this experiment, as well as 10,000 execution-time samples of the generated program to illustrate the variety of possible paths. For the Cortex-M4 setup with enabled instruction cache, the results show an accuracy $\mathcal{A} = \frac{\text{reported WCET}}{\text{actual WCET}}$ of 1.07 for aiT, which is close to the optimum of 1.00. In absolute numbers, aiT overestimates the actual WCET of 68,499 cycles by 4,940 cycles. In contrast, with a reported upper bound of 265,269 cycles PLATIN has an accuracy of 3.87. We attribute this significant overestimation to a great extent to the fact that PLATIN's ARM backend currently assumes a cache miss on every instruction. Repeating the experiment for the Cortex-M4 setup with disabled instruction cache confirms this hypothesis: For this setting, the accuracy of PLATIN improves to 1.77. This is similar to the accuracy of 1.75 PLATIN achieves for Patmos, where both data and instruction cache are enabled, indicating that the PLATIN WCET analyzer models the Patmos platform more accurately than the Cortex-M4 platform. To allow a more comprehensive assessment of the analyzers' accuracies, we repeated the experiment with the identical configuration but varying seed values (1 to 2000). The geometric mean of overestimation factors is 1.47 for PLATIN on the Patmos platform and 1.96/4.01 with de-/activated instruction cache on the Cortex-M4. For aiT on the Cortex-M4 setup, we observed factors 1.23/1.36 with de-/activated instruction cache.



(a) Cortex-M4

(b) Patmos

Figure 4: The WCET value determined by GENE represents the maximum of the execution times of all possible input values.



(a) Cortex-M4 with enabled instruction cache

(b) Cortex-M4 with disabled instruction cache

(c) Patmos

Figure 5: Comparison between actual and reported WCET.

D. Evaluating the Complexity of Benchmarks

Existing benchmark suites contain a significant number of benchmarks that are not resilient against compiler optimizations and/or possess an input-independent control flow [21], as discussed in Section III-B. GENE, on the other hand, ensures that the overall complexity of the generated programs is kept across compiler optimizations and that benchmarks (unless explicitly requested differently) comprise multiple paths. For example, the benchmark used in Section VI-B, which we have executed with all possible input values, on the Cortex-M4 platform leads to paths with 1,998 different execution times ranging from 11,833 to 77,144 cycles with an arithmetic mean of 49,123.85 cycles and a standard deviation of 13,505.65 cycles. These numbers confirm that the inputs to the programs generated by GENE heavily influence the execution times of benchmarks. If necessary, users are able to widen the spectrum of possible paths even further by increasing the probability of conditional-branch patterns being selected and woven into the program during the generation process.

pattern suite	has input?	loops on O0	$\mathcal{R}_{loops}$ O3/O0 [%]	call chain on O0	$\mathcal{R}_{calls}$ O3/O0 [%]	cyclomatic complexity (CC) on O0	$\mathcal{R}_{CC}$ O0/O3 [%]
`simple`	✓	4	100	2	100	353	85
`valueanalysis`	✓	156	100	2	100	822	91
`constantloop`	✗	5	100	2	100	6	100
`inputdependentloop`	✓	12	100	2	100	13	138
`downsamplingloop`	✓	7	100	2	100	15	80
`triangularloop`	✗	408	100	2	100	409	150
`nobranch`	✗	0	–	1	100	1	100

TABLE I: GENE's pattern suites enable the generation of resilient benchmarks with a wide spectrum of different complexities.

To assess the resilience of GENE benchmarks against compiler optimizations, we apply several complexity metrics (i.e., the number of loops, the longest call chain, and the cyclomatic complexity [19]) to a set of benchmarks generated with different pattern suites (see Section IV-D) for a path budget of 20,000. Table I presents the results of this analysis. For all three complexity metrics, we compute the resilience $\mathcal{R} = \frac{\text{measure on O3}}{\text{measure on O0}}$ of a benchmark by comparing the corresponding measures at different optimization levels; a resilience $\mathcal{R}_{loops} = 100\%$, for example, means that a benchmark contains the same number of loops at level `O0` as it does at level `O3`. The optimized versions of the benchmarks are created by LLVM's optimizer `opt`, which as indicated by the complexity measures has a limited influence on programs, but not decisively changes the complexity of benchmarks.

Table I also illustrates that GENE's pattern suites produce benchmarks with a variety of complexities ranging, for example, from 0 to 408 loops. One pattern suite, `nobranch`, constitutes an exception as it has been specifically designed to create simplistic benchmarks used for analyzing the hardware models of WCET tools (see Section VI-E1). Comparing the numbers of Table I with existing benchmark suites for timing analysis shows that for a path budget of 20,000 GENE generates benchmarks that are similar in complexity to TACLEBENCH [11] programs, but significantly more resilient against compiler optimizations [21]. Furthermore, in contrast to existing benchmark suites, for which the complexity of programs is fixed, GENE offers the possibility to increase benchmark complexity by increasing the path budget.

E. Identifying Individual Strength & Weaknesses

Pattern suites allow GENE to generate benchmarks with specific characteristics (see Table I). Below, we use such suites to reveal the individual strengths and weaknesses of aiT and PLATIN with regard to different parts of the timing analysis. In contrast to the experiments presented in previous sections, this time we do not provide the evaluated WCET analyzers with the flow facts of benchmarks in order to examine which program patterns they are able to analyze without assistance.

1) Target-Hardware Analysis: To assess the quality of both WCET analyzers' hardware models, we generate benchmarks with `nobranch`, a pattern suite that produces programs with input-independent control flows consisting of branchless sequences of code, which due to their simplicity put the focus on the target-hardware part of the analysis (see Section IV-D1). For the Cortex-M4 setup with enabled instruction cache, aiT and PLATIN achieve an accuracy of 1.06 and 3.28, respectively. As in the experiment in Section VI-C, the overestimation of the WCET by PLATIN can be attributed to the tool's current pessimistic instruction-cache model. When disabling the instruction cache, the accuracy of PLATIN for the `nobranch` benchmark improves to 2.42. For comparison, for the Patmos architecture PLATIN has an accuracy of 1.08, proving PLATIN's ability to precisely model this platform.

Analyzer	`constant loop`	`input-dependent loop`	`down-sampling loop`	`triangular loop`
aiT	✓	✓	✓	✗
PLATIN	✓	✓	✗	✓

TABLE II: Ability of WCET tools to analyze loop patterns.

2) Value Analysis: GENE provides the pattern suite `valueanalysis` to examine the ability of a WCET tool to analyze value constraints (see Section IV-D2). Benchmarks from this suite mainly consist of arithmetic operations, infeasible paths, and mutually-exclusive paths. When analyzing this benchmark, aiT benefits from its abstract interpretation [32] allowing the tool to determine value constraints of variables, which results in an accuracy of 1.28. PLATIN for the Cortex-M4 setup achieves accuracies of 2.87 (enabled cache) and 1.85 (disabled cache), respectively; both results represent improvements compared to the hardware-model experiment. For Patmos, PLATIN's value analysis has an accuracy of 1.20.

3) Loop-Bound Analysis: In the final experiment, we rely on four different pattern suites, each producing a different challenging loop pattern (see Section IV-D3). Table II presents the results of this experiment. Of the four analysis scenarios, only two are successfully completed by both evaluated WCET analyzers: constant loops and input-dependent loops. In the loop pattern used by the down-sampling loop benchmark, the iteration variable is modified (i.e., decremented) in the loop's body. This modification causes the loop bound to no longer being describable by a closed-form expression. aiT is able to solve this problem with an accuracy of 1.14. PLATIN, on the other hand, fails to bound this loop and would require a manual annotation. In contrast, for the triangular-loop benchmark, aiT is the WCET analyzer that is unable to report an upper bound without assistance, while PLATIN completes the analysis with an accuracy of 2.01. Examining the source code of PLATIN, we found out that the tool for this purpose exploits LLVM's scalar-evolution analysis [33] for loops bounds.

F. Finding Bugs in WCET Analyzers

During our evaluation, most WCET values reported by the WCET analyzers under test, as expected, were overestimations of the corresponding actual WCETs. To our surprise, however, the aiT tool for some GENE benchmarks provided results that were lower than the actual WCETs. Having ruled out that the cause for these invalid WCET values is an error on our side, we contacted AbsInt, the company behind aiT, who confirmed that GENE has discovered a previously unknown bug in the tool. In addition, AbsInt announced to fix this bug as part of a revised version of aiT for the Cortex-M4 platform.

G. Summary

Using the actual WCETs provided by GenE enables us to assess the accuracy of WCET analyzers on an absolute scale. Our results show that the open-source tool Platin accurately models the Patmos architecture, while the commercial tool aiT reports estimates closer to the actual WCET for the Cortex-M4 platform. However, an evaluation with the revised version of aiT resolving the underestimations is part of future work. Both WCET tools need to be provided with flow facts in order to be able to successfully analyze all evaluated program patterns. Otherwise, aiT is unable to bound triangular loops while Platin has problems with down-sampling loops.

VII. Further Usage Scenarios

In this section, we present two further usages of GenE that include generating benchmarks for performance analysis and for workloads used in scheduling analysis.

Generating Benchmarks for Performance Analysis: The field of benchmark generation offers new possibilities to evaluate approaches on a fine-grained level not only for real-time systems. Growing interest exists in generating benchmarks to reveal specific properties [34]. GenE's concept of weaving selectively chosen patterns can be exploited to create benchmarks that stress, for example, the data- or instruction-caching behavior of systems in order to make performance analyses targeting improvements of the average-case execution.

Generating Workloads for Scheduling Analysis: Evaluations of scheduling analysis for real-time systems are carried out increasingly on real hardware platforms using testbeds such as LITMUS^{RT} [35], [36]. For such purposes, De Bock et al. presented a task-set generator, which produces executable binaries with predefined execution times. These binaries are assembled from source code of the TACLeBench suite [11], whereas the main workload is called several times consecutively within the produced code to reach the predefined execution time.

Since GenE is able to generate benchmarks, whose actual WCET is determinable, it is straightforward to generate workloads for scheduling-analysis evaluations. The parameters for the task sets (e.g., WCET, utilization) can be gathered, for example, from the generator of Emberson et al. [37]. GenE is then able to generate benchmarks that lead to these WCET values (e.g., 1 ms). A comprehensive evaluation of this feature for tasks with predefined WCETs and the generation of multi-threaded benchmarks is part of our future work.

VIII. Related Work

GenE is the first benchmark generator that specifically targets evaluating the accuracy of WCET tools by generating benchmarks with known properties (i.e., flow facts, the actual WCET). In addition, this approach proved to be useful for validating the reported upper bounds of these tools.

This paper extends our existing work [38] by the following aspects: definition of flow facts and their mapping between LLVM intermediate representation and machine code (see Section V-A), support to compare analyzers' overestimation factors using two target platforms (see Section VI), several pattern suites (see Section IV-D), and the generation of code with side effects requiring context-sensitive descriptors of patterns and value tracking (see Section V-B).

Lesage et al. [39] proposed a framework for the evaluation of measurement-based timing analyses. Their framework generates abstract-syntax trees (ASTs) comprising abstract tasks. Costs are attributed to the nodes of the AST by running existing application code and deducing basic-block execution models. The generation of tasks is achieved by assembling basic sequences of code, constant loops, and if-elseif-else constructs. The generated AST is simulated; execution times gathered from the simulation are compared against timing analyses on the AST. This approach is beneficial with respect to measurement-based timing analyses operating on an AST level. In contrast, GenE aims at the evaluation of WCET analyzers irrespective of their analysis technique, but with a particular attention to static code analysis. Static code analyzers typically depend on the availability of executable machine code, a requirement their approach unfortunately cannot fulfill. GenE satisfies this dependency by crafting benchmarks with a specific worst-case path leading to the actual WCET, which is used for the evaluation of aiT and Platin.

Several studies exist that discuss the accuracy of the aiT timing-analysis tool [40], [41]. These evaluations suffer from the unavailability of the actual WCET, that is required as a proper baseline for comprehensive evaluations. As a resort, these evaluations compare the reported execution time against measurements gathered from execution traces through the software under analysis. Souyris et al. evaluated aiT by running it on an avionics program [40]. They state that the reported WCET is usually around a factor of 1.25 higher than the execution time gathered from the measurement of the analyzed task. However, their measurements of execution traces are most likely under-estimated values. As a consequence, the comparison between the worst-observed execution time and the reported WCET is of limited expressiveness as it depends on the complexity of the code (e.g., number of mutual exclusive paths, number of value-constrained branches) and the amount of conducted measurements. GenE solves the issue of unknown baselines (i.e., the actual WCET) by generating benchmarks with known properties.

The same problem of unknown baselines exists when comparing timing-analysis tools in the context of the WCET Tool Challenge [12], [13], [14]. The WCET Tool Challenge also provides the source code of several challenging program patterns that, for example, test the ability to determine path-sensitive loop bounds. GenE implements many of these program patterns, which we posed as isolated challenges for Platin and aiT and identified their strengths and weaknesses. Besides these challenging patterns, several benchmark suites exist that are commonly used for benchmarking timing-analysis tools (e.g., [10], [42]). However, since the flow facts of these programs are usually unknown, it is difficult, if not impossible, to extract these facts manually, which are required for determining values close to the actual WCET. GenE solves this issue by generating benchmarks with known flow facts.

IX. Conclusion

The main problem when evaluating WCET tools by using existing benchmarks is that all flow facts and thus the actual WCET is unknown. However, these values are necessary as baseline to determine the accuracy of upper bounds reported by analyzers. In addition to missing baselines, the monolithic structure of benchmarks hides patterns that pose major challenges to tools and lead to overestimations.

To solve these problems, we presented the benchmark generator GenE. The tool builds benchmarks around worst-case paths, which are generated to trigger the actual WCETs under all circumstances. Since GenE operates on a low-level abstraction, it is able to produce code that maintains its configurable complexity across compiler optimizations. Challenging patterns are woven into the program in a context-sensitive way, enabling fine-grained evaluations, for example, of the ability of tools to perform value and loop-bound analyses.

We demonstrated the applicability of GenE on two different hardware platforms. An evaluation of the WCET-analysis tools aiT and Platin confirms that GenE is able to reveal strengths and weaknesses of each analyzer and enabled to detect a bug in aiT causing underestimations of the WCET.

The source code of GenE *is available at:*
`https://gitlab.cs.fau.de/gene`

Acknowledgments: This work is supported by the German Research Foundation (DFG), in part by Research Grant no. SCHR 603/9-2, no. SCHR 603/13-1, and the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89, Project C1).

References

[1] AbsInt. aiT WCET analyzers. https://www.absint.com/ait/.
[2] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard, "The T-CREST approach of compiler and WCET-analysis integration," in *Proc. of SEUS '13*, 2013, pp. 33–40.
[3] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.
[4] B. Lisper, "SWEET – A tool for WCET flow analysis," in *Proc. of ISoLA '14*, 2014, pp. 482–485.
[5] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An open toolbox for adaptive WCET analysis," in *Proc. of SEUS '10*, 2010, pp. 35–46.
[6] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based WCET analysis," in *Proc. of ECRTS '01*, 2001, pp. 37–44.
[7] J. Knoop, L. Kovács, and J. Zwirchmayr, "WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds," in *Proc. of RTNS '13*, 2013, pp. 161–170.
[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of WWC '01*, 2001, pp. 3–14.
[9] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On, "A benchmark characterization of the EEMBC benchmark suite," *IEEE Micro*, vol. 29, no. 5, pp. 18–29, 2009.
[10] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *Proc. of WCET '10*, 2010, pp. 136–146.
[11] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *Proc. of WCET '16*, 2016, pp. 1–10.
[12] N. Holsti et al., "WCET tool challenge 2008: Report," in *Proc. of WCET '08*, 2008, pp. 1–28.
[13] R. von Hanxleden et al., "WCET tool challenge 2011: Report," in *Proc. of WCET '11*, 2011, pp. 1–38.
[14] C. Rochange, "WCET tool challenge 2014," Talk held at WCET '14. [Online]. Available: https://www.irit.fr/wiki/doku.php?id=wtc:start
[15] F. Kluge and T. Ungerer, "EMSBench: Benchmark und Testumgebung für reaktive Systeme," in *Betriebssysteme und Echtzeit*. Springer, 2015, pp. 11–20.
[16] AUTOSAR, "Specification of operating system (version 3.0.2)," Automotive Open System Architecture GbR, Tech. Rep., Jun. 2008.
[17] R. Kirner, A. Kadlec, A. Prantl, M. Schordan, and J. Knoop, "Towards a common WCET annotation language: Essential ingredients," in *Proc. of WCET '08*, vol. 8, 2008, pp. 1–14.
[18] B. Blackham, M. Liffiton, and G. Heiser, "Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets," in *Proc. of RTAS '14*, 2014, pp. 169–178.
[19] T. J. McCabe, "A complexity measure," *IEEE Trans. on Software Engineering*, no. 4, pp. 308–320, 1976.
[20] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, 3rd ed. CRC Press, 2015.
[21] P. Wägemann, T. Distler, P. Raffeck, and W. Schröder-Preikschat, "Towards code metrics for benchmarking timing analysis," in *Proc. of RTSS WiP '16*, 2016, pp. 1–4.
[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of CGO '04*, 2004.
[23] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. of the AMS*, pp. 358–366, 1953.
[24] D.-H. Chu and J. Jaffar, "Symbolic simulation on complicated loops for WCET path analysis," in *Proc. of EMSOFT '11*, 2011, pp. 319–328.
[25] P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat, "Worst-case energy consumption analysis for energy-constrained embedded systems," in *Proc. of ECRTS '15*, 2015, pp. 105–114.
[26] G. Bernat, R. Davis, N. Merriam, J. Tuffen, A. Gardner, M. Bennett, and D. Armstrong, "Identifying opportunities for worst-case execution time reduction in an avionics system," *Ada User Journal*, vol. 28, no. 3, pp. 189–195, 2007.
[27] J. Abella et al., "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *Proc. of SIES '15*, 2015, pp. 1–10.
[28] B. Huber, D. Prokesch, and P. Puschner, "Combined WCET analysis of bitcode and machine code using control-flow relation graphs," in *Proc. of LCTES '13*, 2013, pp. 163–172.
[29] ARM Limited, "Cortex-M4 technical reference manual," 2010.
[30] Infineon Technologies AG, "XMC4500 reference manual," 2012.
[31] M. Schoeberl et al., "T-CREST: Time-predictable multi-core architecture for embedded systems," *JSA*, vol. 61, no. 9, pp. 449–471, 2015.
[32] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand, "An abstract interpretation-based timing validation of hard real-time avionics software," in *Proc. of DSN '03*, 2003, pp. 625–632.
[33] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of Recurrences – a method to expedite the evaluation of closed-form functions," in *Proc. of ISSAC '94*, 1994, pp. 1–8.
[34] H. Borghorst, K. Bieling, and O. Spinczyk, "Towards versatile models for contemporary hardware platforms," in *Proc. of OSPERT '16*, 2016.
[35] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proc. of RTSS '06*, 2006, pp. 111–126.
[36] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, UNC Chapel Hill, 2011.
[37] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. of WATERS '10*, 2010, pp. 6–11.
[38] P. Wägemann, T. Distler, T. Hönig, V. Sieh, and W. Schröder-Preikschat, "GenE: A benchmark generator for WCET analysis," in *Proc. of WCET '15*, 2015, pp. 31–40.
[39] B. Lesage, D. Griffin, F. Soboczenski, I. Bate, and R. I. Davis, "A framework for the evaluation of measurement-based timing analyses," in *Proc. of RTNS '15*, 2015, pp. 35–44.
[40] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, and G. Borios, "Computing the worst case execution time of an avionics program by abstract interpretation," in *Proc. of WCET '05*, 2005, pp. 21–24.
[41] C. Ferdinand, R. Heckmann, H.-J. Wolff, C. Renz, O. Parshin, and R. Wilhelm, "Towards model-driven development of hard real-time systems," in *Proc. of ASWSD '06*, 2006, pp. 145–160.
[42] N. Holsti, T. Langbacka, and S. Saarinen, "Using a worst-case execution time tool for real-time verification of the DEBIE software," in *Proc. of DASIA '00*, 2000, pp. 1–6.