To appear in: Proceedings of the 15th International Workshop on Worst-Case Execution Time Analysis
Lund, Sweden, 7 July 2015

GenE: A Benchmark Generator for WCET Analysis

Peter Wägemann, Tobias Distler, Timo Hönig, Volkmar Sieh, Wolfgang Schröder-Preikschat

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Abstract

The fact that many benchmarks for evaluating worst-case execution time (WCET) analysis tools are based on real-world applications greatly increases the value of their results. However, at the same time, the complexity of these programs makes it difficult, sometimes even impossible, to obtain all corresponding flow facts (i.e., loop bounds, infeasible paths, and input values triggering the WCET), which are essential for a comprehensive evaluation. In this paper, we address this problem by presenting GenE, a benchmark generator that in addition to source code also provides the flow facts of the benchmarks created. To generate a new benchmark, the tool combines code patterns that are commonly found in real-time applications and are challenging for WCET analyzers. By keeping track of how patterns are put together, GenE is able to determine the flow facts of the resulting benchmark based on the known flow facts of the patterns used. Using this information, it is straightforward to synthesize the accurate WCET, which can then serve as a baseline for the evaluation of WCET analyzers.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases WCET, benchmark generation, flow facts, WCET Tool Challenge

Digital Object Identifier 10.4230/OASIcs.WCET.2015.31

1 Introduction

Benchmarks such as Mälardalen [11], DEBIE-1 [13], or PapaBench [19] play an important role in the evaluation of WCET analyzers as they allow to assess the individual strengths and weaknesses of different tools, for example, in the context of the WCET Tool Challenge. If the focus of such an evaluation lies on the comparison of two or more WCET analyzers, having the source code and/or binaries of a benchmark available is usually sufficient to achieve meaningful results. However, for a comprehensive investigation, additional questions about the properties of the program under test need to be answered in order to be able to objectively assess the quality of a particular WCET analyzer:

1. What are the loop bounds?
2. What are the feasible paths?
3. What are the recursion depths?
4. What are the concrete input values triggering the WCET?

Throughout this paper, we refer to such information as the flow facts of a benchmark. Their significance stems from the fact that flow facts can serve as an absolute baseline for the validation of WCET-analyzer results, thereby enabling evaluations that go beyond the relative comparison of multiple tools. Note that depending on the analysis approach used, some flow facts are more important than others: while loop bounds, feasible paths, and recursion depths are especially relevant in the context of a static timing analysis, having available concrete input values triggering the WCET, for example, is of major importance for measurement-based timing analysis. With knowledge about the concrete worst-case input

15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015).
Editor: Francisco J. Cazorla; pp. 31–40

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

values, which also take hardware features (e.g., caching) into account, the actual WCET of a benchmark can be determined by concretely executing it on a cycle-accurate simulator or on the specific target platform while measuring its execution time.

Unfortunately, extracting all possible flow facts from existing benchmarks is inherently difficult, in some cases even impossible: From a theoretical point of view, non-trivial properties of program code are not automatically decidable [22]. This applies also to existing benchmark suites [11, 13, 19] written in high-level programming languages.

Knowledge about all existing flow facts requires knowledge about all existing program paths, which makes it necessary to explicitly enumerate these paths. Such an explicit path enumeration is infeasible for most real-world programs due to the myriad of possible paths [17]. Furthermore, manually determining all possible flow facts of programs is labor-intensive and error-prone [3].

In this paper, we address the problem of determining flow facts for WCET benchmarks by presenting GenE[1], a tool that provides both WCET benchmarks as well as their flow facts. GenE is able to achieve this, because instead of analyzing existing programs, the tool generates new benchmarks by relying on small building blocks (for which the flow facts are known) and combining them in a way that allows the tool to automatically determine the flow facts of the resulting complex benchmark. In order to create realistic benchmarks, the building blocks used by GenE are typical design and implementation patterns that have been extracted from existing real-world applications and WCET benchmark suites.

In particular, the contributions of this paper are:

- GenE, a tool to automatically generate benchmarks, enabling users to conduct comprehensive evaluations of WCET analyzers.
- An algorithm to create complex WCET benchmarks from program patterns that allows to keep track of flow facts.
- A discussion of how the flow facts provided by GenE can be used to determine the WCET of a generated benchmark for both static and measurement-based timing analysis.

The remainder of this paper is structured as follows: Section 2 discusses a number of general principles by which the evaluation of a WCET analyzer should be guided. Section 3 presents details of GenE and explains how our tool enables users to meet these principles. Section 4 provides an overview of related work, and Section 5 concludes and gives an outlook on future work.

2 A Vision for the Comprehensive Evaluation of WCET Analyzers

Conducting a comprehensive evaluation of WCET analyzers is a challenging task. In the following, we identify a number of requirements that are crucial in this context and discuss whether, and if so to which extent, they are met by existing approaches.

Large Number of Benchmarks. In order to get stable and comparable results, WCET analyzers should be evaluated using a large set of different benchmarks. This minimizes the risk that a few poorly-selected benchmarks lead to false conclusions.

In practice, the evaluation of WCET analyzers is usually performed based on an existing benchmark suite [11, 12, 19, 20] or a set of manually-selected programs. As a consequence, the number of benchmarks used typically ranges between one and around twenty. While

[1] The name GenE originates from the term genie and the idea of **Gen**erating **E**valuation sets.

this may be sufficient for some use cases, in general it is desirable to have a greater number of benchmarks. Considering the effort it takes to select/implement a benchmark, we argue that this goal can only be achieved by generating benchmarks automatically.

Realistic Benchmarks. WCET analyzers should be evaluated under real-world conditions. In consequence, in the optimal case, the benchmarks used are actual applications or at least closely resemble them.

Many benchmark suites available have been composed with a focus on real-world programs and consequently meet this requirement. Of course, this is also implicitly true for evaluations that are performed based on actual applications.

Wide Spectrum of Benchmarks. An evaluation should take into account that different applications have different characteristics, in particular, with regard to complexity and program structure. For example, while state-machine-like applications are usually built around a central control loop, signal-processing applications in contrast often consist of a static input array and nested loops. Introducing variety into the set of benchmarks offers the possibility to properly reflect such differences. Furthermore, it allows to better compare the individual strengths and weaknesses of the WCET analyzers evaluated.

The state-of-the-art Mälardalen WCET benchmarks [11] and the TCAS benchmark [9], which was used at the WCET Tool Challenge 2014, cover both of the categories mentioned above: state-machine-like applications (i.e., `statemate`, `tcas`) as well as signal-/image-processing programs (i.e., `fir`, `edn`, `jfdctint`).

Preservation of Benchmark Properties. The selection procedure must take into account that compiler optimizations may have significant impact on programs. As a result, the timing analysis either needs to be performed on an already optimized representation of the benchmark or needs to ensure that the benchmark properties are preserved by the compiler across code transformations.

Having investigated several publicly-available benchmarks, we found that without further modifications a significant number of Mälardalen WCET benchmarks is not resistant to compiler optimizations, by default, without setting additional preprocessor definitions: For example, Clang optimizes five of these benchmarks (i.e., `bs`, `expint`, `fibcall`, `janne_complex`, `ns`) to a single return statement. This is due to values stored in variables that have no effect on the output enabling compilers to decisively change the structure of the program.

Availability of Benchmark Flow Facts. In order to be able to independently assess the quality of a WCET analyzer, the flow facts of a benchmark should be known prior to the evaluation. This way, a WCET analyzer can be rated based on how close its outputs are to the actual results of a benchmark.

In general, there is a tradeoff between the demand to have realistic benchmarks and the requirement to obtain flow facts, as precise numbers are usually not available for real-world applications for reasons of complexity (see Section 1). As a consequence of the fact that, as discussed above, many existing benchmarks focus on realistic use cases, evaluations are therefore mostly based on a comparison of results, or rely on an absolute baseline that is either determined manually or an over-approximation [3, 17]. That is, in order to evaluate the impact of a novel feature, the WCET analysis is performed twice: once with the feature enabled and once with the feature disabled. Although such an approach allows to assess the relative improvement achieved by the feature, it is not suitable to determine the benefit of the feature on a global scale.

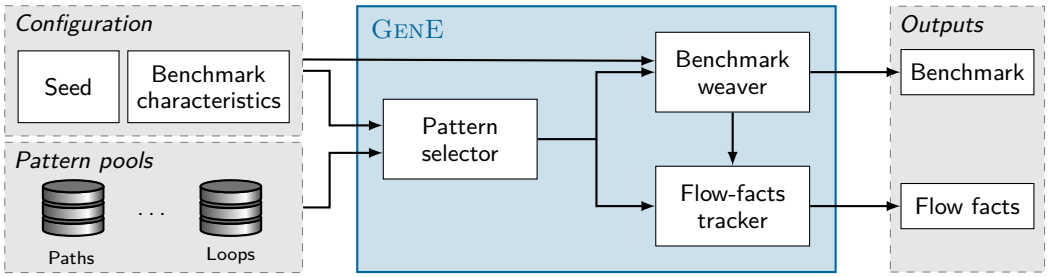

Figure 1 GenE automatically generates benchmarks by combining existing program patterns.

3 GenE

In this section, we present GenE, an approach and tool to address the problem of providing benchmarks for the evaluation of WCET analyzers. GenE generates benchmarks automatically and therefore allows a **large number of benchmarks** to be used, as they no longer have to be selected or implemented manually. In order to provide **realistic benchmarks**, the tool combines common building blocks (e.g., conditional statements and nested loops) we extracted from real-world applications. By varying the selection and composition of patterns, GenE supports the creation of a **wide spectrum of benchmarks**. To assemble a new program, GenE relies on a deterministic procedure designed to ensure the **preservation of benchmark properties**. In addition, this procedure also allows the tool to determine the flow facts of a benchmark based on knowledge about its structure, enabling GenE to guarantee the **availability of benchmark flow facts** for all programs created.

Figure 1 shows an overview of the GenE workflow: When a user requests the tool to create a benchmark with certain characteristics (e.g., those of a signal-processing application), the tool first selects suitable programs patterns from a set of pattern pools (see Section 3.1). In the next step, GenE composes the actual benchmark by weaving the selected patterns into a program (see Section 3.2). Based on the information which patterns have been selected as well as knowledge about how they have been put together, the tool is then able to track the flow facts for the generated benchmark (see Section 3.3).

3.1 Patterns and Pattern Selection

When generating a new benchmark, the first step performed by GenE is to automatically select different building blocks, which in a later step are then combined to a complex program. In order to enable the tool to create realistic benchmarks, we conducted a study of existing programs used for the evaluation of WCET analyzers to identify recurring patterns. The results of this analysis are a combination of design and implementation patterns described in literature [7] as well as patterns directly extracted from the code of state-of-the-art benchmark suites such as Mälardalen [11]. Listings 1 and 2 present two simple examples of patterns used by GenE: a path pattern including a conditional branch and a parameterizable nested-loop pattern. As shown in the listings, both patterns comprise a number of *insertion points*, which allows GenE to combine them with other patterns during the benchmark-weaving step (see Section 3.2).

GenE manages the building blocks of benchmarks in a set of pattern pools, thereby grouping together patterns with similar structure (e.g., array accesses in loops). Users are able to implement new patterns and add them to existing pools; in addition, the tool also offers the possibility to introduce new pattern pools. Pattern pools play an important

Listing 1 Example of a path pattern

```
if( condition ){
  // insertion_point_1
} else {
  // insertion_point_2
}
// insertion_point_3
```

Listing 2 Example of a loop pattern

```
for(i = n-1; i >= 1; i--){
  for(j = 0; j < i; j++){
    // insertion_point_1
  }
}
// insertion_point_2
```

role during the pattern-selection step: By assigning different weights to different pools, GenE ensures that the benchmark generated matches the properties specified by the user. For example, if a user requests a program matching the characteristics of a digital-signal-processing application, the tool favors the selection of loop patterns operating on arrays, as such patterns are typical for this category of use cases.

3.2 Benchmark Weaving

After the patterns have been selected, they are combined during the benchmark-weaving phase to create new complex benchmarks. To address the problem of tracking flow facts (see Section 3.3), GenE uses a formal grammar G that is considered by the weaving algorithm. The grammar G contains the start symbol (S) and the empty string (ε). A point in the control-flow graph of the generated benchmark where further expansions through statements (e.g., assignments, loops, branches) are possible is called an insertion point (inp) (see Listing 1 and 2). The following list is an excerpt of the production rules of G:

1. $\mathcal{S} \mapsto FunctionBegin \cdot inp \cdot FunctionEnd$
2. $inp \mapsto \varepsilon$
3. $inp \mapsto Statement \cdot inp$
4. $Statement \mapsto Assignment$
5. $Statement \mapsto IfBegin \cdot inp \cdot ElseBegin \cdot inp \cdot EndIf$
6. $Statement \mapsto LoopHead \cdot inp \cdot LoopEnd$

⋮ *(Additional production rules)*

The start symbol is mapped to exactly one insertion point (1), which is surrounded by the begin and end of the main function. Each insertion point can produce an empty element (2) or a statement followed by a further insertion point. Production rule 3 ensures the sequential creation of patterns. The nonterminal *Statement* symbols can produce assignments, branches, or loops (4–6).

The grammar describes what benchmarks can be produced whereas the weaving algorithm presented in Listing 3 exemplarily illustrates how GenE uses this grammar to create a benchmark. The weaving algorithm recursively inserts new patterns into insertion points. The core concept of the algorithm is to bound the timing cost for inserted patterns from top to bottom. There can be program paths through this pattern that are less expensive than the worst-case path[2], but the worst-case path must exactly lead to the predefined cost. All operations in Listing 3 based on a seed value are prefixed with `select_`, concrete insertions of code into the current selected insertion point are prefixed with `emit_`, and the `costof()` function returns the cost of sequential statements. While descending the tree, the costs are subdivided and used for further productions. The algorithm receives a list of available

[2] It is assumed that the worst-case path is defined only through its concrete input values and no other effects like concurrency or non-deterministic input/output operations.

Listing 3 The recursive algorithm of GenE inserts patterns top-down into the benchmark.

```
1  function GenE(vars, cost)
2   if(cost == 0) return vars
3
4   switch(select_production(cost))
5     case Statement · inp:
6        statement_cost ← select_cost ∈ [0, cost]
7        new_vars ← GenE(vars, statement_cost)
8        new_vars ← GenE(new_vars, cost - statement_cost)
9     case Assignment:
10       (new_vars, operation) ← select_assignment(vars)
11       emit_Assignment(operation)
12    case IfBegin · inp · ElseBegin · inp · EndIf:
13       condition ← select_condition(vars)
14       emit_IfBegin(condition)
15       if_cost ← cost - costof(condition) // if-case is worst case here
16       else_cost ← if_cost - (select_cost ∈ [0, if_cost])
17       if_vars ← GenE(vars, if_cost)
18       emit_ElseBegin()
19       else_vars ← GenE(vars, else_cost)
20       emit_EndIf()
21       new_vars ← merge_variables(if_vars, else_vars)
22    case LoopHead · inp · LoopEnd:
23       ...
24  return new_vars // return updated variables including value ranges
```

variables and the timing cost that must be produced by the emitted code for the respective call of the function (Line 1). If no cost is available for productions, the algorithm returns the current variables including their values (Line 2), which is the termination criterion for the algorithm. Otherwise, a production is selected based on the distributable cost (Line 4).

Cost Distribution. The cost is subdivided into parts based on the seed. The parts are used for generating a statement followed by an insertion point (Line 5–8). The concrete cost modeling, which is necessary for determining worst-case paths including their input values, is discussed in Paragraph 3.3.1.

Recursive Application. After determining the cost for the statement, GenE is recursively applied (Line 7) with the respective cost. The rest of the distributable cost is used for a second recursive call of the GenE function enabling sequences of pattern insertions.

Value Tracking. For computed values that are used, for example, for branch conditions, the values of all variables and their availability must be tracked. For example, the *Assignment* production can introduce a new variable that is computed from two further variables (Line 10). Consequently, the updated variables are returned at the current insertion point during the code-generation process.

3.3 Flow-Facts Tracking

Tracking the value ranges of variables is an essential mechanism for the flow-facts tracking that takes place during the benchmark-weaving phase. These flow-facts include, for example, feasible paths, loop bounds, or values triggering the most expensive path. The concrete input value leading to the WCET is determined prior to the first call of the GenE function based on the seed value. Consequently, all costs of feasible branches in the control flow must be modeled according to the input variable taking computations on the input into consideration.

Reconsidering Listing 3, when selecting the branch condition during the application of the *If-Else* production (Line 13), the current values of the used variables are taken into

account and the costs of the branches (Line 15–16) are set according to the worst-case input. The cost of each branch is again exactly bounded through the recursive calls of GenE (Line 17, 19). The possible values of the variables are merged after emitting the *If-Else* pattern and the values for the worst-case path are stored (Line 21).

The patterns hold their own (parameterizable) flow facts, for example, as formulas that must be determined when the pattern is manually constructed. These formulas are considered when inserting the pattern and combined with the existing flow facts. For example, reconsidering the parameterizable loop pattern shown in Listing 2, the loop bound of the outer loop is `n-1` whereas the parametric formula for the inner body is `n*(n-1)/2`. These formulas are used when the pattern is woven into the code and the concrete flow facts are determined with the concrete value of the variable `n`.

3.3.1 Cost Modeling

WCET analysis is typically split into two parts: a high-level analysis of flow facts and a low-level analysis of the processor (i.e., instruction execution times, cache modeling). GenE focuses on generating benchmarks with challenging, high-level flow facts and not on difficult access patterns for cache analysis for a specific architecture. However, to distribute cost for patterns, knowledge on timing costs of the target platform is necessary. GenE addresses the cost-modeling problem through *relatively* modeling timing costs [23] and *overweighting* the branches of determined worst-case paths with these relative costs of instructions.

Reconsidering the *If-Else* path pattern shown in Listing 1, the branch of the *If* case is overweighted by a large factor. This factor for overweighting worst-case branches must ensure that even if the *Else* branch only creates cache misses, the *If* branch is still more time consuming. Furthermore, since GenE is implemented on a low-level intermediate representation of program code (see Section 3.4), this representation can be attributed with concrete cost information [6, 10] to further refine the cost model through target-specific knowledge allowing smaller factors for overweighting the worst-case path.

However, for comparing the precision of WCET-analysis tools, an absolute WCET value must be known for the generated path conditions. A discussion of how the flow facts provided by GenE can be used to determine the WCET is described in the following paragraph.

3.3.2 Determination of the WCET

Detailed knowledge about the input values triggering the worst-case path is stored through the value and flow-facts tracking. Consequently, the generated benchmark can be concretely executed with the determined worst-case input leading to the actual WCET when measuring the time of this execution. The concrete execution and precise measurement can be achieved through two approaches. First, modern processors are equipped with highly accurate internal time-stamp counters that allow precise time measurements without the need of external measurement hardware. Second, if a cycle-accurate simulator for the targeted processor is available, the actual WCET can be derived through a simulation, without requiring the target hardware. Once the actual WCET of the generated program is determined, it serves in combination with the known flow facts as a baseline for the evaluation of both static and measurement-based timing analyzers. Precisely modeling the behavior of the processor is not mandatory in such an input-oriented approach as long as the determined input values lead to the WCET when executing the program with them, which is achieved through the relative overweighting of branches. Consequently, complex hardware features (e.g., caching) are implicitly respected through the concrete execution.

3.4 Implementation

GenE is implemented with the framework of the Low Level Virtual Machine (LLVM) [18] using its intermediate representation (LLVM-IR) for the following reasons:

- The generated benchmarks are independent from specific processors to generate machine code for various targets. The cost modeling is performed on basis of the LLVM and is further improvable through target-specific knowledge [6, 10].
- Although the LLVM-IR is independent of specific target machines, it can be precisely mapped to machine code through control-flow relation graphs [14]. Using such graphs, the knowledge of flow facts can be transformed from the LLVM-IR level to machine code.
- If the language of the generated benchmarks is implemented on a higher abstraction level, it is more difficult to track the control-flow relations during optimizing source-to-source transformations. However, GenE focuses on generating benchmarks that look like optimized code, since such code facilitates timeliness and unoptimized code is unrealistic for real-world scenarios. To handle this problem, GenE uses optimized patterns that are further combined. During lowering the LLVM-IR to machine code, the generated benchmark must not be optimized further and must not change its control flow.

GenE implements several challenging program patterns taken from [7, 11] and is extensible through the integration of additional patterns. Further benchmark suites, where patterns with different characteristics are extractable for weaving new benchmarks, are MiBench [12], BEEBS [20], or PapaBench [19].

To evaluate the effectiveness of our benchmark-generation algorithm, we run GenE with different initial seeds and maximum distributable cost leading to different execution times. With this configuration, GenE is able to generate 10,000 benchmarks within four minutes on an Intel Core-i7 (8 GB RAM). Although using a single integer input value with 32 bits, 2^{32} control flows through the generated benchmark can be produced. This input space is considered to be sufficiently large for WCET benchmarking.

4 Related Work

GenE is the first approach to automatically generate benchmark programs specifically targeting the evaluation of WCET-analysis tools. However, the development of this benchmark generator is strongly inspired by the Csmith [24] benchmark generator. Csmith targets the domain of testing compilers for their correctness, for example, through generating complex branch expressions. The concept of inserting patterns into existing code recursively top-down and thereby creating new insertion points is similar to the recursive expansion process of Dujmović [8] for generating performance benchmarks. Additionally to this approach, GenE tracks low-level flow facts and worst-case input values targeting the WCET domain. In the domain of high-performance computing, the MicroProbe framework [2] is an automated approach to micro-benchmark generation. The generated benchmarks are, for example, utilized for energy-related characterization of a specific target architecture. We consider benchmarks generated by GenE as suitable source for comparing the effectiveness of existing WCET-analysis tools [1, 5, 16, 21].

To compare best the flow facts found by WCET analyzers, GenE must support to output flow facts in a format that analyzers under evaluation can process, which is considered future work. An example for such a flow-fact language is FFX [4]. Further discussions on suitable annotation languages to provide flow facts are found in [15].

5 Conclusion and Outlook

Comparing WCET-analysis tools through benchmarking is a challenging task since many aspects must be considered, such as providing a wide spectrum and large number of benchmarks for which the flow facts are available. We therefore propose the GenE benchmark generator that is able to automatically generate benchmarks with diverse characteristics including their flow facts. With knowledge about these flow facts and input values, a precise WCET value can be determined of the generated benchmark, which serves as a common baseline for timing-analysis tools. Therefore, we consider to use benchmarks built by GenE together with generated flow facts, the input assignment, and the WCET as a suitable evaluation scenario for upcoming WCET Tool Challenges, in addition to existing benchmarks.

Especially for the upcoming challenges in multi-core WCET analysis of complex concurrency patterns, the GenE approach is useful to create a known baseline. This is due to the fact that these benchmarks are more complex compared to benchmarks on single-core processors. Consequently, manually determining the WCET of multi-core benchmarks is inherently more labor-intensive than of single-core benchmarks. An incremental process is imaginable where new challenging concurrency patterns are progressively integrated into GenE and WCET tools catch up with the analysis of these patterns. To improve the existing prototype of GenE and to make it applicable for use, GenE will be made available after incorporating feedback from the WCET community.

Acknowledgements This work is supported by the German Research Foundation (DFG), in part by Research Grant no. SCHR 603/9-1 (Aspect-Oriented Real-Time Architecture), Research Grant no. SCHR 603/13-1 (Power-Aware Critical Sections), and the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89, Project C1).

References

1 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

2 R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzalez, and P. Bose. Systematic energy characterization of CMP/SMT processor systems via automated micro-benchmarks. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 199–211, 2012.

3 B. Blackham, M. Liffiton, and G. Heiser. Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets. In *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium*, pages 169–178, 2014.

4 A. Bonenfant, H. Cassé, M. De Michiel, J. Knoop, L. Kovács, and J. Zwirchmayr. FFX: A portable WCET annotation language. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 91–100, 2012.

5 A. Bonenfant, M. de Michiel, and P. Sainrat. oRange: A tool for static loop bound analysis. In *Proceedings of the Workshop on Resource Analysis*, 2008.

6 C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Proceedings of the 17th International Symposium on Low Power Electronics and Design*, pages 333–338, 2011.

7 D.-H. Chu and J. Jaffar. Symbolic simulation on complicated loops for WCET path analysis. In *Proceedings of the 9th International Conference on Embedded Software*, pages 319–328, 2011.

8 J. Dujmović. Automatic generation of benchmark and test workloads. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 263–274, 2010.

9 A. Gotlieb. TCAS software verification using constraint programming. *The Knowledge Engineering Review*, 27(03):343–360, 2012.

10 N. Grech, K. Georgiou, J. Pallister, S. Kerrison, and K. Eder. Static energy consumption analysis of LLVM IR programs. *Computing Research Repository, arXiv*, pages 1–12, 2014.

11 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pages 137–147, 2010.

12 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 *Proceedings of the International Workshop on Workload Characterization*, pages 3–14, 2001.

13 N. Holsti, T. Langbacka, and S. Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. In *Proceedings of the Data Systems in Aerospace Conference*, pages 1–6, 2000.

14 B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proceedings of the 14th Conference on Languages, Compilers and Tools for Embedded Systems*, pages 163–172, 2013.

15 R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET analysis: The annotation language challenge. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis*, pages 1–17, 2007.

16 J. Knoop, L. Kovács, and J. Zwirchmayr. r-TuBound: Loop bounds for WCET analysis. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 435–444, 2012.

17 J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 161–170, 2013.

18 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

19 F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. PapaBench: A free real-time benchmark. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*, pages 1–6, 2006.

20 J. Pallister, S. J. Hollis, and J. Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *Computing Research Repository, arXiv*, pages 1–12, 2013.

21 P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *Proceedings of the 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 1–8, 2013.

22 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, pages 358–366, 1953.

23 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 *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 1–10, 2015.

24 X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, pages 283–294, 2011.