Appears in: Proceedings of the 37th Real-Time Systems Symposium Work-in-Progress Session (RTSS WiP 2016)

Towards Code Metrics for Benchmarking Timing Analysis

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

***Abstract*—The use of standardized programs is a necessary means for benchmarking the effectiveness of worst-case execution time (WCET) analysis tools and enables a comparison between different tools. For comprehensive evaluations of these tools and their methods, a characterization of benchmark properties is inevitable, as it indicates the potential analysis complexity for WCET tools. A challenging problem is the fact that complexity measures are not necessarily stable across different program representations from source to machine code, since modern compilers fundamentally transform control-flow structures due to aggressive optimizations. Additionally, selecting an unsuitable set of metrics potentially causes misleading complexity measures.**

In our ongoing work, we develop a tool that gathers complexity measures from code, such as numbers of (nested) loops and paths or call-stack depths. We also exploit a method to automatically reveal data-flow–independent control flows, to identify benchmarks that are unsuitable for WCET benchmarking. The tool correlates these complexity numbers across different levels of optimization and enables a combined interpretation of applied metrics. As part of our evaluations, we applied the metrics on the suite TACLeBench to investigate which benchmarks are resilient against compiler optimizations and maintain challenging properties for benchmarking WCET analyzers.

I. Introduction

Worst-case execution time (WCET) analysis tools are crucial for determining upper bounds on execution times of scheduled tasks in real-time systems. To verify the effectiveness of these tools and their implemented methods as well as to enable a comparison between different WCET tools, a variety of benchmarks is available [1], [2], [3], [4]. In order to conduct a comprehensive evaluation that reveals the full potential of WCET analyzers, it is important to select benchmarks that due to their complexity pose a challenge to the tools. This selection process usually involves the use of metrics [5], [6] yielding complexity measures such as the number of loops, function calls, or linearly independent paths in a benchmark program, following the rationale that such elements in general make WCET analyses more difficult.

In this paper, we identify two major pitfalls associated with the selection of WCET benchmarks based on complexity metrics. *Pitfall 1: The measures provided by such metrics are not necessarily stable across compiler optimizations.* As a result, a benchmark that based on a certain metric is complex at the source-code level might end up trivial (according to the same metric) at the optimized machine-code level where the WCET tool performs its analysis. Due to compiler techniques such as loop unrolling and function inlining, typical examples of complexity metrics affected by this problem are the numbers of loops or function calls contained in a program.

Pitfall 2: Interpreting measures in isolation can lead to misleading assessments of the difficulty of benchmarks. This is especially true for benchmarks whose control flows are independent of their input-data values and which consequently always execute the same path. Determining the execution times of such benchmarks is usually straightforward even though other measures (e.g., a large number of loops and/or function calls) might suggest the opposite.

To address these problems that arise in the context of selecting WCET benchmarks, we are currently developing a tool to facilitate the selection process. In particular, our tool automatically analyzes benchmark programs at multiple code levels, thereby tracking and comparing measures across different degrees of compiler optimizations. As a result, the tool is able to evaluate the resilience of programs against optimizations, allowing users to identify and consequently discard benchmarks that only appear complex at the source-code level but in fact do not pose a challenge to WCET analyzers. Instead of interpreting measures in isolation, our tool relies on a combination of several metrics (e.g., number of loops, number of paths, and inputs) and incorporates additional analyses for the detection of data-independent control flows in order to enable a comprehensive evaluation of WCET tools.

The paper is structured as follows: Section II elaborates on the pitfalls associated with using metrics for WCET analysis. Section III describes our approach towards code metrics for benchmarking timing analysis as well as the prototype implementation of our tool. Section IV presents evaluation results, Section V discusses related work, and Section VI concludes.

II. Problem Statement

In the following, we analyze different metrics typically used for categorizing WCET benchmarks in order to identify potential pitfalls associated with them.

A. Loops, Paths, and Function Calls Across Optimizations

The number of loops, paths, and function calls usually varies in programs on different levels of compiler optimization. The problem when using such metrics is discussed as follows.

1) Loops: One of the most common metrics used for quantifying the complexity of a WCET benchmark is the existence and/or number of loops in a program, following the assumption that automatically determining loop bounds is a challenging task for WCET tools. However, while this might be true for nested loops in which the number of iterations of the inner loop depends on the iteration variable of the outer

loop, there are many loops at the source-code level that do not pose a problem to WCET analyzers, mainly because the tools do not see them when conducting their analyses on optimized machine code. Listing 1 shows an example of such a loop where the loop bound is fixed through the bounded size of the input data. As a consequence, compilers are able to fully unroll the loop, resulting in optimized code that no longer contains the loop header and thus is easier to analyze (see Listing 2).

Listing 1. Unoptimized, bounded loop

```
#define INPUT_SIZE 4
for(i=0; i<INPUT_SIZE; ++i){
  work(i);
}
```

Listing 2. Unrolled loop

```
work(0);
work(1);
work(2);
work(3);
```

2) Number of Paths: Static WCET analysis considers paths from a given control-flow graph as constraints for the analysis. Therefore, the number of paths in the program indicates the complexity of the benchmark. When reconsidering the code examples in Listing 1 and Listing 2, similar problems as with loops under optimizations arise for the number of paths.

The cyclomatic complexity introduced by McCabe [6] provides a software metric that yields the number of linearly independent code paths through a control-flow graph. For a single function, the cyclomatic complexity M can be formulated as $M = E - N + 2$, where E denotes the number of edges and N the number of nodes in the control-flow graph. Depending on the representation of the loop as control-flow graph of Listing 1, the cyclomatic complexity is at least $M = 2$. As a result of compiler optimization, the original complexity caused by branches in the loop header is reduced due to the loop unrolling and the loop pattern degraded to a straight-line sequence of code ($M = 1$).

3) Number of Function Calls: The code in Listing 2 can potentially be further transformed by an optimizing compiler: the instructions behind function `work()` can be inlined into the caller function in order to increase performance. This means that inlining changes the number of function calls making the number-of-calls metric as optimization-level–dependent as the metrics number-of-loops or number-of-paths.

4) Conclusions: Based on the observations discussed above we draw the following conclusions: First, metrics such as the number of loops or paths can provide significantly different values depending on the code level at which they are applied; consequently, it is important to always specify the corresponding code level alongside the actual measures. Second, when selecting benchmarks for the evaluation of a WCET analyzer, it is crucial to especially take the code level into account at which the analysis is actually conducted. Third, WCET-tool evaluations should preferably rely on those benchmarks whose complexity is not reduced by compiler optimizations.

B. Isolated Interpretation of Complexity Measures

Describing the complexity of benchmarks by applying isolated metrics can be misleading, requiring a combined application of metrics. This holds especially for measures that are necessary preconditions for challenging benchmarks, such as the existence of data-dependent control flows.

1) Number of Loops versus Number of Function Calls: Besides actually reducing complexity (e.g., by loop unrolling), compilers potentially apply transformations that shift complexity from one complexity measure to another. A typical example for such a transformation is function inlining. Functions that are subject of inlining potentially contain loops. During inlining, new independent loops of the same shape are inserted into the code and, as a consequence, the number of loops increases while the number of calls decreases. In this example, the benchmark maintains its overall complexity for WCET analysis, although the number-of-loops measure increases.

2) What are Inputs to Programs? An understanding of input data used for benchmark programs is crucial for timing analysis. The MÄLARDALEN benchmark suite [1] provides a notion of input data [7]: if variables (i.e., function parameters, external variables) exist that affect the control flow, the benchmark is not a single path and consequently has an input. Otherwise, the program has a data-flow–independent control flow. Data-flow–independent control flows are common for bounded input sizes. For example, encryption algorithms are usually agnostic to actual input-data values and therefore implement single-path functions. These data-flow–independent control flows are straightforward to analyze[1]: When the input does not affect the control flow, WCET tools are able to perform an end-to-end simulation of the single path through the program. Independent from the input values, the program always has the same execution time, which is problematic for benchmarking timing analysis tools.

3) Conclusions: We draw two main conclusions from the above-mentioned observations: First, compiler optimizations potentially shift complexity from one measure to the other. As a consequence, an isolated interpretation of one metric might lead to false conclusions when the complexity is hidden behind another measure. Second, combined application of metrics is especially important with a detection of data-flow–independent control flows. If a benchmark contains only a single path, the program is straightforward to analyze by WCET tools.

III. Approach Towards WCET Metrics

The main advances of our approach, which measures complexity on different optimization levels and enables a combined interpretation of metrics, are presented in the following.

A. Measures Across Optimization Levels

To avoid the pitfall of unstable complexity measures across compiler optimizations, our approach considers different optimizations levels. From the comparison of complexity numbers between unoptimized (source) code and highly-optimized (machine) code, we formulate measures on the resilience against compiler optimizations: the *resilience factors* $\mathcal{R}_i$. For each metric (e.g., number of loops), we formulate a distinct resilience factor (e.g., $\mathcal{R}_{loops}$). For example, a program with N loops in source code and zero loops in machine code is likely to be unsuitable for benchmarking, since the factor indicates

[1]We assume that, for WCET benchmarking, architectures are used without unbounded timing anomalies and that execution traces are deterministic.

$\mathcal{R}_{loops} = \frac{0}{N} = 0\,\%$ resilience against optimizations. Another example for the necessity of measures across levels is that values of $\mathcal{R}_{loops} \geq 100\,\%$ indicate that new loops of equal shape were generated due to function inlining.

B. Combining Metrics & Interpreting Measures

Isolated interpretations of complexity measures potentially lead to false conclusions. To avoid this pitfall, our approach considers a combined application of different metrics. Specifically, our approach considers seven distinct analyses:

1) Detection of inputs (i.e., data-dependent control flows)
2) Detection of recursion
3) Detection of function-pointer usages
4) Detection of floating-point usages
5) Longest call chain
6) Number of loops including their nesting depths
7) Cyclomatic complexity (number of independent paths)

The steps of assessing the suitability of a benchmark for WCET benchmarking is sketched in the following: The existence of data-flow–dependent control flows is a necessary precondition for a suitable benchmark. Next, metrics with low resilience factors are considered as weak. However, this only applies if a low resilience of one metric (e.g., R_{loops}) is not compensated by another metric (e.g., R_{calls}), since complexities can be shifted from one metric to another, because of optimizing control-flow transformations (see Section II-B). Benchmarks with reports of unboundedness and comparatively high complexity measures indicate suitable benchmarks.

C. Implementation

The current prototype of our tool for building complexity measures is implemented as distinct analysis passes inside the LLVM compiler framework, a framework that targets highly optimized code. Our analyses run on the target-agnostic intermediate representations of LLVM. This setup provides the possibility to gather complexity measures of benchmarks written in many high-level programming languages. Besides the source code, an entry point (i.e., function name) for the analysis is required, which enables analyses from arbitrary program points while omitting unwanted initialization code.

Measures possibly report unboundedness: for example, when recursive function calls are detected, the longest call chain cannot be derived statically in the general case.

Our analyses make conservative assumptions. Two examples outline these conservative assumptions: First, when a cycle in the call graph is detected, the benchmark is marked as recursive. However, call-graph cycles are a necessary, not a sufficient, condition for recursiveness. Second, the detection of data-flow–dependent input is not straightforward to achieve if programs use pointers. As a consequence, our detection algorithm in such cases relies on information from alias analyses to determine whether an input affects the control flow. If the result from the alias analysis indicates potential aliasing, our algorithm conservatively marks this benchmark as input dependent. Further advanced context-sensitive analyses and refinements are subject of future work.

benchmark	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 [%]	use float?
adpcm_dec	✓	6	0	3	100	32	16	✗
adpcm_enc	✓	7	14	4	75	39	28	✗
ammunition	✓	77	155	∞	∞	1465	102	✗
anagram	✓	19	137	∞	∞	74	107	✗
audiobeam	✓	19	58	7	57	121	88	✓
basicmath	✓	11	145	5	80	329	108	✓
binarysearch	✓	1	200	2	50	4	125	✗
bitcount	✓	4	125	∞	∞	20	105	✗
bitonic	✓	2	100	∞	∞	10	90	✗
bsort	✓	2	150	2	50	6	67	✗
cjpeg_transupp	✓	56	66	2	100	73	86	✗
cjpeg_wrbmp	✓	5	140	3	67	55	71	✗
complex_updates	✗	1	100	1	100	2	100	✓
countnegative	✗	2	100	2	50	4	75	✗
cover	✗	3	100	2	50	194	95	✗
crc	✓	3	100	3	67	19	100	✗
dijkstra	✓	5	120	3	67	19	84	✗
duff	✗	0	–	2	50	10	10	✗
epic	✗	40	110	5	100	211	236	✓
fac	✓	1	300	∞	∞	4	200	✗
fft	✗	6	100	2	100	9	122	✗
filterbank	✓	14	93	2	150	15	113	✓
fir2dim	✗	13	15	2	100	22	14	✓
fmref	✓	11	209	6	67	167	107	✓
g723_enc	✓	6	117	5	80	83	98	✗
gsm_dec	✓	17	141	7	71	119	111	✗
gsm_encode	✓	48	96	7	71	250	102	✗
h264_dec	✓	13	38	2	100	123	8	✗
huff_dec	✓	12	167	4	75	38	150	✗
huff_enc	✓	20	135	∞	∞	100	116	✗
iir	✗	1	0	1	100	2	50	✓
insertsort	✓	2	100	1	100	7	114	✗
jfdctint	✗	2	100	2	100	3	100	✗
lift	✓	6	50	5	60	60	70	✗
lms	✗	5	60	2	100	6	67	✓
ludcmp	✓	9	122	3	100	15	140	✓
matrix1	✗	3	67	1	100	4	75	✗
md5	✓	9	133	7	71	62	100	✗
minver	✓	17	82	3	100	33	103	✓
mpeg2	✓	33	106	6	83	531	68	✗
ndes	✓	12	92	4	100	29	214	✗
petrinet	✓	1	100	1	100	126	100	✗
pm	✓	30	103	3	133	67	143	✓
powerwindow	✓	6	67	5	100	168	80	✗
prime	✓	1	200	4	25	8	138	✗
quicksort	✓	10	180	∞	∞	55	135	✓
recursion	✓	0	–	∞	∞	5	80	✗
rijndael_dec	✓	8	75	3	100	34	88	✗
rijndael_enc	✓	10	70	3	100	39	79	✗
sha	✓	17	59	5	80	80	99	✗
st	✓	4	225	4	25	23	113	✓
statemate	✓	1	100	3	100	183	63	✗
susan	✓	52	96	6	83	813	83	✓
test3	✗	121	100	22	100	705432	100	✗

TABLE I
EXCERPT OF COMPLEXITY MEASURES OF TACLEBENCH

IV. EVALUATION

We evaluate our tool on a preliminary version (September 2016) of the benchmark suite TACLEBENCH [2]. All analyses start at the entry-point function annotated in each benchmark. Table I shows an excerpt of complexity numbers. In this table, a checkmark indicates a positive answer to a posed question and the symbol ∞ indicates unboundedness of analysis results.

The suite contains benchmarks with a huge range of complexity measures. Regarding the reduction of complexity, 19 of the total 54 benchmarks (35 %) are not resilient against compiler optimizations and lose loops due to loop unrolling. For example, the complexity of the program `adpcm_dec` is

radically reduced by these optimizations: it contains 6 loops on level `O0`. After optimizations, all loops are unrolled and none remain on level `O3` (i.e., $R_{loops} = 0\,\%$).

The number of, for example, loops can actually increase from `O0` to `O3` (see values above 100 % in column 3). This is because of function inlining where loops modularized into functions on level `O0` are directly inserted into the optimized code, which then creates new loops of equal shape. For example, in `basicmath`, the number of loops increases by 45 %. However, our tool reveals that the longest call chain decreases from 5 to 4 ($R_{calls} = 80\,\%$), which is an indicator for function-call inlining. The combined consideration of measures reveals that the complexity was shifted from one measure to another and avoids the pitfall of interpreting measures in isolation.

The first column of Table I shows results from the detection of data-dependent control flows. In summary, 12 of the 54 benchmarks (22 %) consist of a data-flow–independent control flow and consequently implement a single-path program. Amongst these is benchmark `test3`, which is the prime example for the necessity of combined metrics: the `test3` benchmark contains the maxima for longest call chain (22), number of loops (121), and a cyclomatic complexity (705432) of all benchmarks. Additionally, it has a strong resilience against compiler optimizations: no call is inlined ($R_{calls} = 100\,\%$), no loop is unrolled ($R_{loops} = 100\,\%$), and no paths are removed ($R_{CC} = 100\,\%$). However, the program has no data-flow–dependent control flow and consequently its WCET, which is equal to the best-case execution time, is analyzable by a trace through the single path, which is not challenging to analyze for WCET tools.

V. Related Work

The MÄLARDALEN WCET suite [1] provides a categorization of benchmarks that includes measures indicating whether a benchmark uses include files, calls external library routines, is a single-path program, contains loops, contains nested loops, uses arrays and/or matrices, uses bit operations, contains recursion, contains unstructured code, or uses floating-point calculations; it also reports the size of the source-code file and the lines of source code. In our tool we reuse most of these metrics that are applicable on LLVM intermediate representation. Additionally, we compare measures across different optimization levels to expose the resilience of benchmarks against compiler optimizations.

Audsley et al. [8] proposed a framework to assess code quality of automatic code-generator tools. In their evaluations, they apply metrics such as lines of code or the cyclomatic complexity using the tool CCCC [9]. To evaluate the complexity of code for WCET analysis, further metrics such as number-of-loops, inputs, or call chains are of interest as well as a correlation of these numbers on highly optimized code.

The EEMBC benchmark suite [3] provides a characterization of properties with the focus on performance analysis. The programs are structured by their usage of functional units in the processor (e.g., load-store unit, arithmetic logic unit). For benchmarking timing analysis, these values are not sufficient, since the challenging aspects are problems such as the detection of loops or value constraints. However, adding such categorizations for WCET benchmarking is considerably helpful for the low-level hardware-related part of the analysis.

VI. Conclusion & Future Work

We developed a tool that applies several code metrics and respects the impact of compiler optimizations. As a result, the tool yields measures that indicate the resilience of benchmarks against optimizations. Furthermore, the detection of inputs and combined interpretation of metrics enable an identification of programs that are suitable for WCET-tools benchmarking.

Other large-scale benchmark suites, such as the LLVM test suite [10] containing ten thousands of source-code lines, demand for an automated approach to assess properties. We assume that running our tool on this suite will reveal further potential for WCET benchmarks. These suitable benchmarks can then be integrated into standardized suites.

Our metrics tool currently operates at the granularity of entire benchmark programs. Building on this functionality, as future work we will focus on an automatic identification of challenging program patterns that are known to be complicated for WCET tools [11]. The determined patterns will then be integrated into our benchmark generator GENE [12].

The source code of our prototype to gather measures from code is publicly available: `gitlab.cs.fau.de/gene/`

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

References

[1] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *Proc. of the 10th Intl. Work. on Worst-Case Execution Time Analysis*, 2010, pp. 136–146.

[2] 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 the 16th Intl. Work. on Worst-Case Execution Time Analysis*, 2016, pp. 1–10.

[3] 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.

[4] 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 the Intl. Work. on Workload Characterization*, 2001, pp. 3–14.

[5] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, 3rd ed. CRC Press, 2015.

[6] T. J. McCabe, "A complexity measure," *IEEE Trans. on Software Engineering*, no. 4, pp. 308–320, 1976.

[7] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. (2016) (discussion on input variables). http://www.mrtc.mdh.se/projects/wcet/inputs.html.

[8] N. Audsley, I. Bate, and C. O'Halloran, "An assessment framework for automatic code generator tools," in *Proc. of the 22nd Intl. System Safety Conf.*, 2004, pp. 665–674.

[9] T. Littlefair. (2016) CCCC – C and C++ code counter. http://cccc.sourceforge.net/.

[10] LLVM test suite. (2016) LLVM testing infrastructure guide. http://llvm.org/docs/TestingGuide.html.

[11] D.-H. Chu and J. Jaffar, "Symbolic simulation on complicated loops for WCET path analysis," in *Proc. of the 9th Intl. Conf. on Embedded Software*, 2011, pp. 319–328.

[12] P. Wägemann, T. Distler, T. Hönig, V. Sieh, and W. Schröder-Preikschat, "GenE: A benchmark generator for WCET analysis," in *Proc. of the 15th Intl. Work. on Worst-Case Execution Time Analysis*, 2015, pp. 1–10.