Early WCET Prediction Using Machine Learning

Armelle Bonenfant[1], Denis Claraz[2], Marianne de Michiel[3], and Pascal Sotin[4]

1 **University of Toulouse, IRIT, Toulouse, France**
bonenfant@irit.fr
2 **Continental Automotive, Toulouse, France**
denis.claraz@continental-corporation.com
3 **University of Toulouse, IRIT, Toulouse, France**
michiel@irit.fr
4 **University of Toulouse, IRIT, Toulouse, France**
sotin@irit.fr

Abstract

For delivering a precise Worst Case Execution Time (WCET), the WCET static analysers need the executable program and the target architecture. However, a prediction – even coarse – of the future WCET would be helpful at design stages where only the source code is available. We investigate the possibility of creating predictors of the WCET based on the C source code using machine-learning (work in progress). If successful, our proposal would offer to the designer precious information on the WCET of a piece of code at the early stages of the development process.

1998 ACM Subject Classification D.4.7 Real-Time Systems and Embedded Systems

Keywords and phrases Early WCET, Machine Learning, Static Analysis, C Language

Digital Object Identifier 10.4230/OASIcs.WCET.2017.5

1 Introduction

Context and Motivation

The Worst Case Execution Time (WCET) static analysers operate – for most of them [10, 7, 3, 12] – on the binary of the program under analysis, taking into account the target architecture. With these elements the WCET estimate can be proven safe and might hopefully be precise. Unfortunately these requirements make that the WCET estimate is available late in the development process when many choices linked to the WCET have already been done.

▶ **Situation 1.** *At Continental Automotive, any new SW project version is based on already existing components on the shelf, as well as newly developed components. The existing components usually have associated WCET (obtained by measurement) for some architectures. For the components that have no timing attached or only timings for too distinct architectures, a tool operating on source code and providing a prediction of the future WCET would be of great help. In particular in multi-core context, an early estimation of the runtime would help the integrator to choose the right core for integration.*

Work In Progress

We are currently investigating the possibility of applying machine learning to obtain source-code-based WCET predictors (specific to the compilation chain and architecture on which they were learnt). In this article:

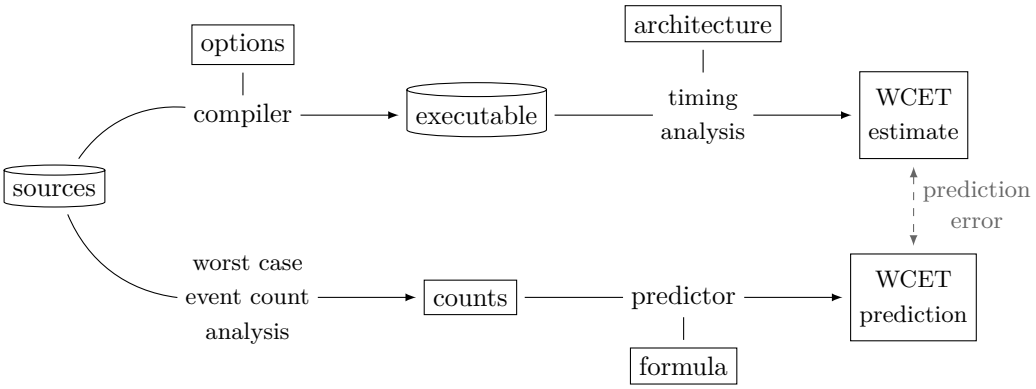

Figure 1 Two ways to approach the WCET.

- We introduce a framework for creating the so-called predictors (Section 2). This framework rely on *static analysis* of the source code, on *machine learning* and on a set of programs which final WCET estimate is known or can be computed.
- We present a novel source code static analysis that counts certain events occurring in the worst case execution (Section 3).
- We discuss the application of machine learning in our setting (Section 4). In particular, we list several questions that have not received satisfying answers yet.

2 Overview

In some industrial settings, as illustrated in Situation 1, the size of the project (e.g. 1.5 M effective lines of code) or the complexity of the compilation chain makes it time consuming to create the program binary and thus to obtain the WCET. In addition, the company organisation might imply that distinct persons from distinct departments provide the source code, integrate and build the binary. Due to the size of the projects, and the short development times, a parallelization of the developments is mandatory, and therefore iterative integration cycles are done. When doing an integration step, having an idea of the future WCET of a given component (newly developed, or old but not measured nor analysed) is important since it might influence the selection of the core, the position in the task, or in worst case, the whole project architecture. Early availability of the information is here more important than accuracy of the result, the effective real time behaviour being tested extensively or statically analysed in a further phase. This need of an early WCET is also mentioned by [11, 6, 1].

Our current work pursues the idea that it might be possible to get a satisfying prediction of the future WCET by applying a formula on some characteristics of the source code (Figure 1). This formula has to be learnt from a set of programs which WCET estimate is already known (Figure 2).

In Figure 1, we depict two ways of getting an idea of the WCET.

- The top line is the usual WCET derivation, we call its result *WCET estimate*. This estimate is by construction an over-approximation of the WCET.
- The bottom line is our proposal. We call its result a *WCET prediction*. This prediction cannot be safely used as WCET estimate but should give an relatively close approximation of the top line WCET estimate if we can find a satisfying formula.

A similar pattern is followed by [6, 1] but relies on measurements (related work is detailed in Section 5).

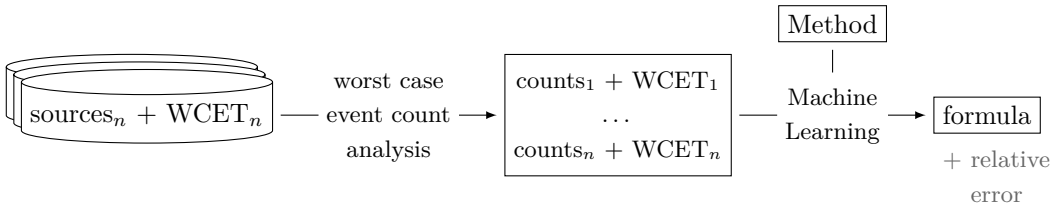

Figure 2 Learning WCET source code predictors.

Discovering a formula linking precisely enough the WCET to the source code for a given compilation chain and architecture is the main goal of our work. In order to achieve this goal we put our faith in machine learning.

First of all, we note that machine learning cannot be applied straightforward on programs. The best it can deliver us is to link the WCET estimate with discrete or continuous attributes of the program source code. We identified two classes of attributes that can be fed to the machine learning:

1. *Count* attributes. For example the maximal number of arithmetic operations, function calls or global variable read accesses. These attributes give a concrete idea of the program duration and could be assembled by the learning tool into a nice formula.
2. *Style* attributes. For example the number of lines of code, the maximal loop nesting, whether the program was generated or hand-written. These attributes are by themselves insufficient to get an idea of the program duration but they can help the learning tool to define classes of programs.

We defined a static source code analysis that we call *worst case event count analysis*. It extracts count attributes (Item 1) from the source code. This analysis provides more than source-code-based WCET analysis [9] since it delivers counts for several categories of events and not a single already-aggregated WCET estimate. In our process, aggregation is left to machine learning. Details are given in Section 3.

The worst case event count analysis is performed in two situations:

- Before applying the formula on a given source code (bottom line of Figure 1),
- Before feeding a set of programs to the machine learning process (first step of Figure 2).

Figure 2 depicts the learning process. This process starts with a set of programs for which the WCET estimates are known. The only assumption that we make on these WCET estimates is that they intend to reflect the WCET of the source code compiled for a given architecture. The worst case event counts are then extracted for these programs and paired with the WCET estimates. The spreadsheet obtained is then fed to the machine learning tool that will output a formula predicting the WCET estimate in function of the event counts. Depending on the method used by the tool the formula can range from a linear combination of the events to a neural network.

In Section 4 we discuss the problems we face in that learning process.

3 Count Attributes Extraction

In this section, we present how we extract worst case event counts from the C source code. Our proposal is linked with source code timing scheme [9] but the data manipulated are more complex (count by categories and sets of such metrics).

(a) X is smaller than Y in each category

(b) X and Y are not ordered as in Figure 3a but an optimistic Δ_Y cost more than a pessimistic Δ_X

Figure 3 Two reasons for ignoring metric X in front of metric Y.

3.1 Analysis Principles

Our analysis is an evaluation of the source code syntax, aware of the loop bounds.

Analysis Domain

We consider an `if-then-else` statement where the `then` and `else` branch evaluations gave respectively the metrics T and E such that:

$$T = [\mathit{Arith} \mapsto 10; \mathit{Load} \mapsto 3], \qquad E = [\mathit{Arith} \mapsto 2, \mathit{Load} \mapsto 5]\,.$$

Where *Arith* and *Load* respectively stand for *performing a basic arithmetic operation* and *read access to a variable*. What should then be the evaluations of the `if-then-else` statement? An imprecise but "sound" solution would be to use the smallest metric greater than T and E, also known as the least upper bound $T \sqcup E$ of T and E:

$$\begin{aligned} T \sqcup E &= [\mathit{Arith} \mapsto \texttt{max}(10, 2); \mathit{Load} \mapsto \texttt{max}(3, 5)] \\ &= [\mathit{Arith} \mapsto 10; \mathit{Load} \mapsto 5]\,. \end{aligned}$$

This option is appealing because it seems to mimic the over-approximation that will be performed by the timing analysis. This is not the case. When the binary WCET analysis has to select the `then` or the `else` branch it reasons on numbers of cycles that are totally ordered. The solution it retains might really be the worst case. In our case, the metric $T \sqcup E$ is wrong: no "execution" can reach these figures. Using the average or the minimum instead of the maximum would not deliver a more satisfying result.

The metrics are structured as a partially ordered set (poset) isomorphic to $\mathbb{N}^{\mathbf{c}}$ where $\mathbf{c}$ is the set of event categories. A precise solution is to to keep track of *sets of maximal metrics*. For the `if-then-else` statement above the evaluation would be the set $\{T, E\}$ since none of them is greater than the other. Potentially an evaluation could contain as many elements as the number of permutations on $\mathbf{c}$ ie. $|\mathbf{c}|!$. In practice the size of the set does not grow that fast and we use some architectural knowledge in order to state for example that:

$$[\mathit{Load} \mapsto 5; \mathit{Store} \mapsto 2] \prec [\mathit{Load} \mapsto 40; \mathit{Store} \mapsto 1]\,.$$

The argument that sustains this ordering is the fact that one extra store cannot be more expensive than 35 extra loads. More formally, each category receives a pessimistic and an optimistic evaluation in terms of cycles. With these ranges we define the two functions $\mathit{eval}_{\text{pess}}$ and $\mathit{eval}_{\text{opt}}$ that assign a numerical value to a metric. We then redefine the order on the metrics as follows:

$$X \prec Y \iff \mathit{eval}_{\text{opt}}(X - Z) < \mathit{eval}_{\text{pess}}(Y - Z) \qquad \text{with } Z = X \sqcap Y\,.$$

Table 1 Categories used by our worst case event count analysis.

Family	Category	Optimistic	Pessimistic
Operations	Simple	0.5	1
	Multiplication	1	5
	Division	1	10
Control	Unconditional branch	0.5	1
	Conditional branch	1	1
	Computed branch	1	4
	Call	0.5	10
Memory	Address setting	0.5	1
	Load	2	20
	Store	2	20

where the metric $X \sqcap Y$ is the greater lower bound of X and Y (it is computed by keeping the minimum for each category) and the metric $X - Z$ is the difference of X and Z, category by category. This ordering on metrics encompasses the point-wise ordering. Figure 3 can give the visual intuition of this ordering.

Eventually, the set of maximal metrics computed for the whole program needs to be brought back to a single metric in order to be fed to the learning process. At that stage there is often few elements R_i in the set because when numbers of events get large, the ordering eliminates many metrics that cannot be the worst-case path. We approximate to $R_1 \sqcup \ldots \sqcup R_n$.

Analysis Rules

The sets of maximal metrics ($\mathcal{A}$, $\mathcal{B}$) are combined according to these rules:

$\mathcal{A}$ and $\mathcal{B}$ are in sequence	$\uparrow \{X + Y \mid X \in \mathcal{A}, Y \in \mathcal{B}\}$,
$\mathcal{A}$ and $\mathcal{B}$ are alternatives	$\uparrow (\mathcal{A} \cup \mathcal{B})$,
$\mathcal{A}$ is repeated n times	$\{n \times X \mid X \in \mathcal{A}\}$,

where the metric $X + Y$ is the sum of X and Y, category by category, the metric $n \times X$ is the multiplication of each category of X by n and the set $\uparrow \mathcal{S}$ is the set of metrics $\mathcal{S}$ where non-maximal elements have been removed.

3.2 Implementation

We implemented the analysis presented in Section 3.1 on top of the oRange static analysis tool for C [8]. Table 1 shows the considered categories and associated optimistic and pessimistic evaluation for events of these categories (in cycles).

4 Machine Learning

In this section, we present our machine learning environment (Section 4.1), discuss the difficulties we face setting up the framework of Figure 2 (Section 4.2) and present our plans for the near future (Section 4.3).

4.1 Machine Learning Methods

We will use WEKA [5], a machine learning software. It is a collection of state-of-the-art visualization tools and algorithms for data analysis and predictive modelling. Among other techniques, it supports data preprocessing, clustering, classification and regression.

Once data has been collected on a large set of programs, the software will provide us classifications, and ways to visualize our data. It allows the systematic comparison of the predictive performance of WEKA's machine learning algorithms on a collection of datasets.

WEKA can be set up to compute the error while learning: the data is divided in 10 packs; in turn, 9 are used to learn and remaining one is used to compute the error.

The validity of the learnt formula depends on the quantity and the quality of the data.

4.2 Learning WCET Predictors

Program Set

In order to have a precise learning, the program set should be large and representative – preferably thousands of elements. Finding such a set of programs is definitely an issue:

- Benchmarks, like the TACLeBench [4], seldom contain more than a hundred programs. These programs are representative but heterogeneous (size, style, purpose).
- Program generators, like CSMITH [13], offer as many programs as needed. These programs might be far from representative. In addition, their lack of meaning makes that clever compiler may optimize them in unpredictable proportions (dead code elimination, pre-computation) thus breaking any correlation between the source and the binary.
- Industrial program base, like the one mentioned in Situation 1, may contain hundred of software components. These program are representative and homogeneous.

The set of programs used should also be correlated with the categories of event considered. Events that are absent or simply invariant in the program set might be integrated to the formula in an absurd way.

Choice of the Learning Method

The WEKA machine learning application offers a *huge* number of learning methods, some of them presenting tricky parameters. The exploration of this possibilities is a non-trivial dimension of our problem.

Validation of the Predictor

As mentioned in Section 4.1, the machine learning algorithms already offer a notion of relative error. When considering large programs, we would consider as successful a relative error of $\pm 20\%$.

If the chosen method delivers us a weight for each event category we can validate that:

- The weights are within the optimistic/pessimistic ranges used in Section 3.
- The weights are sensible values for a WCET expert.

Relevant Event Categories

The categories listed in Table 1 where chosen because they seem relevant from a WCET point-of-view. For example, multiplication and division have been set aside from the other operations because their compilation can range from a shift to an emulation function.

Some of these categories can be shown to be useless by our future work while some might need to be refined. If our proposal is successful, the final categories could be considered as an interesting production.

Feasibility

Eventually, we need to consider this hypothesis:

▶ **Hypothesis 2.** *The formula we are looking for does not exists.*

For a given compilation process and micro-architecture Hypothesis 2 means that it is impossible to correlate any source code characteristics to the WCET of the binary. This hypothesis is not absurd since the impact of the compilation and the hardware on the WCET can be huge and far from regular. This hypothesis cannot be proven because failing to learn can be the consequence of either nothing to learn or bad learning method.

If Hypothesis 2 holds, then we propose two ways to fight against it:

- Make a step toward the binary by performing the front-end compilation passes, then applying the worst case event count on the intermediate representation. Hopefully these passes will be common to most compilation chains.
- Restrict our claim to a family of programs.

4.3 Roadmap

A modified version of the static analyser oRange [8] provides us the worst case event count analysis. The WCET analysis framework Otawa [2] will be used to compute the WCETs of the program base. The machine learning will be provided by the Weka platform.

We have proceeded to a first experiment in order to get a proof of concept. We did the experiment on random generated programs. The learnt formulae were too complex and the imprecisions were too wide. We believe that this failure is due to the use of programs having WCET of different magnitudes. We tried to learn from programs being too different.

Thus, for this work, our roadmap is as follow:

- Getting a random program generator in order to calibrate the programs. We want to get a classification prior to submit to Weka: program having similar size, similar structure (loops/no loop, conditions/no condition, nested/non nested loops, number of event...). At first, we used CSmith [13] but we need more calibration possibilities.
- For each category of programs, getting a formula
- Combining the class of programs and getting a new valid formula
- Classifying the programs of a benchmark (or use case) and running the formula.

5 Related Work

In [6, 1], Gustafsson et al. address the same problem with a different tool. From a set of measurement on several programs, they infer a timing model. Once learnt, this timing model can be applied to the measurements of a new piece of code. The technique used to learn the model is a variation of a linear regression adapted to the fact that only some of the measurements might reflect worst case behaviours. This technical point prevents the exploration of the learning techniques we can afford since it would lead to learning an average behaviour. A important distinction between this work and our proposal is that our proposal requires no access to the effective hardware but an already set up static analysis

tool (e.g. OTAWA and the hardware model) while their work requires no pre-existing model but an access to the execution platform.

As already mentioned in Section 2 and in the header of Section 3 our proposal offer a feature similar to [9] with the difference that we transfer the task of aggregating cleverly the software events to a machine learning tool. Our proposal is thus more complex (need worst case event count analysis) but we benefit from machine learning and we can (try to) learn any architecture.

6 Conclusion

Early WCET evaluation is a recurrent request in the domain. Our approach face two main issues: finding relevant attributes and applying relevant machine learning analysis.

Our expertise on classical WCET evaluation by static analysis is solid. We think we are able to identify and compute qualitative attributes for C programs.

The second issue will obviously require machine learning expertise in addition to time analysis expertise. Once the programs are classified, we are confident that we will be able to obtain a calculus for early WCET. Applying this approach to general programs is more ambitious, our study will tell the conditions of applications.

Acknowledgements. We wish to thank Mathieu Serrurier for patiently sharing with us bits of its expertise in machine learning.

References

1 Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, 2016. doi:10.1007/s11241-016-9250-7.

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems – 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.

3 Franck Cassez and Jean-Luc Béchennec. Timing analysis of binary programs with UPPAAL. In Josep Carmona, Mihai T. Lazarescu, and Marta Pietkiewicz-Koutny, editors, *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*, pages 41–50. IEEE Computer Society, 2013. doi:10.1109/ACSD.2013.7.

4 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASIcs*, pages 2:1–2:10. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

5 Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, Bernhard Pfahringer, Ian H. Witten, and Len Trigg. Weka-a machine learning workbench for data mining. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook, 2nd ed.*, pages 1269–1277. Springer, 2010. doi:10.1007/978-0-387-09823-4_66.

6 Jan Gustafsson, Peter Altenbernd, Andreas Ermedahl, and Björn Lisper. Approximate worst-case execution time analysis for early stage embedded systems development. In Sunggu Lee and Priya Narasimhan, editors, *Software Technologies for Embedded and Ubiquitous Systems, 7th IFIP WG 10.2 International Workshop, SEUS 2009, Newport Beach, CA, USA, November 16-18, 2009, Proceedings*, volume 5860 of *Lecture Notes in Computer Science*, pages 308–319. Springer, 2009. doi:10.1007/978-3-642-10265-3_28.

7 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In Richard Gerber and Thomas J. Marlowe, editors, *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS 1995). La Jolla, California, June 21-22, 1995*, pages 88–98. ACM, 1995. doi:10.1145/216636.216666.

8 Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008. doi:10.1109/RTCSA.2008.53.

9 Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991. doi:10.1109/2.76286.

10 Peter P. Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989. doi:10.1007/BF00571421.

11 David Trilla, Javier Jalle, Mikel Fernández, Jaume Abella, and Francisco J. Cazorla. Improving early design stage timing modeling in multicore based real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 305–316. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461338.

12 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

13 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.