Identifying Relevant Parameters to Improve WCET Analysis*

Jakob Zwirchmayr[1], Pascal Sotin[1], Armelle Bonenfant[1], Denis Claraz[2], and Philippe Cuenot[2]

1 **Université de Toulouse, IRIT, France**
2 **Continental Automotive France SAS, Toulouse, France**

Abstract

Highly-configurable systems usually depend on a large number of parameters imposed by both hardware and software configuration. Due to the pessimistic assumptions of WCET analysis, if left unspecified, they deteriorate the quality of WCET analysis. In such a case, supplying the WCET analyzer with additional information about parameters (a scenario), e.g. possible variable ranges or values, allows reducing WCET over-estimation, either by improving the estimate, or by validating the initial estimate for a specific configuration or mode of execution. Nevertheless, exhaustively specifying constraints on all parameters is usually infeasible and identifying relevant ones (i.e. those impacting the WCET) is difficult. To address this issue, we propose the branching statement analysis, which uses a source-based heuristic to compute branch weights and that aims at listing unbalanced conditionals that correspond to system parameters. The goal is to help system-experts identify and formulate concise scenarios about modes or configurations that have a positive impact on the quality of the WCET analysis.

1998 ACM Subject Classification B.2.2 Performance Analysis and Design Aids, C.3 Special-Purpose and Application-Based Systems, C.4 Performance of Systems

Keywords and phrases WCET Accuracy, Modes and Configuration, Flow Facts, Scenario Specification

Digital Object Identifier 10.4230/OASIcs.WCET.2014.93

1 Introduction

At Continental Automotive France SAS and in the automotive industry in general, the reuse of software is a major source of quality improvement and development cost reduction. Such reuse is enabled by product-platform approaches, for example by offering a clear decoupling between generic function development and application project integration. Therefore, the corresponding software modules are firstly developed in a platform context which makes them applicable in a large diversity of contexts, such as engine cylinder number, combustion type or fuel type. Abstraction of the system configuration and the hardware platform results in a configurable software module solution, for which the worst-case execution time (WCET) depends on a series of parameters. Additionally, software functions must be frequently adapted to changes of the configuration or enhanced by new features. Besides scheduling analysis and timing constraint verification, WCET analysis is applied in various phases of the development cycle of such an industrial project. In the design phase, it is used to estimate the computation power required by a brand new functionality in order to properly size the

* This work is supported by ANR W-SEPT.

hardware configuration. During module development, WCET analysis is applied to verify the compliance to certain platform rules, like, e.g., the maximum interrupt blocking time.

Due to a huge amount of unspecified settings in such a configurable environment the pessimistic assumptions of WCET analysis usually leads to a high over-estimation of the WCET, especially when it is applied in an early development phase.

Supplying precise information about possible values of relevant parameters can thus improve the quality of the WCET estimate or establish that the WCET estimate for the particular configuration coincides with the reported unconstrained WCET estimate. Manually identifying and specifying constraints on all relevant parameters is a tedious task. In order to assist such a task we propose an approach, *branching statement analysis*, that focuses on identifying crucial branching choices at control flow level. To this end, we identify parameters by analyzing and listing conditionals that are deemed WCET-relevant by our analysis. Then, a system-expert provides additional information about the particular configuration in form of input constraints (scenario). WCET analysis of the system under the supplied scenario either leads to an improvement of the WCET estimate for the analyzed mode, or validates the accuracy of the (global) analysis for this particular configuration.

The contribution of this work is an approach that makes the task of identifying relevant parameters in a configurable system less tedious, while gaining on the precision of the WCET estimate reported for the configuration. Our method relies on a static analysis tool to compute loop bounds and infeasible path information and can be applied iteratively to incorporate and refine value specifications of parameters. By relying on a source based timing heuristic, we identify *unbalanced* conditionals and guide the system-expert when constructing and/or refining short and precise scenario specifications about relevant parameters.

The rest of the paper is structured as follows. An overview of the approach is presented in Sect. 2, followed by an industrial point of view on WCET analysis in Sect. 3. A detailed description of our method is presented in Sect. 4, while Sect. 5 overviews effects of scenarios on the WCET analysis precision. We report on experiments in Sect. 6 followed by an overview of related work, Sect. 7, and we conclude in Sect. 8.

2 Motivating Example

The diagram shown in Figure 1 outlines the setup of our approach. Traditionally, to compute a WCET estimate, a (possibly empty) scenario and the program source code are supplied to a control flow analyzer (`oRange` [7]) that computes flow facts about the program (FFX [10]). The program binary and the flow facts are supplied as input to the low-level analyzer (`OTAWA` [1]) that computes a WCET estimate for the program. We extended the flow fact computation step by a Δ-computation step. The goal is to find unbalanced conditionals (in terms of weight, currently a syntactical measure) related to parameter values, such that a system-expert can focus on specifying parameters that are deemed relevant by the branching statement analysis. Specifying data constraints for those parameters yields a refined scenario that can again be supplied to the WCET computation step. The analysis can be re-run iteratively using the refined scenario.

As stated in Section 1, the WCET estimate usually depends upon a large number of parameters imposed by outside constraints. Therefore, one often is interested in estimating the WCET of the program in a specific mode of execution or configuration.

In this paper, we denote as the *configuration* of the system the hardware and software environment of a component. A *mode of execution* describes running the program under certain assumptions about the environment. *Parameters* then denote variables that reflect

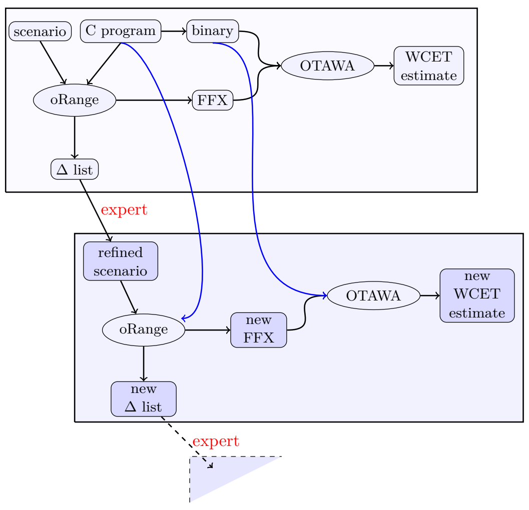

Figure 1 Introducing Δ-values to support scenario specification and refinement.

```
#include "missing.h"
int expensive() { /* ... */ }
int cheap() { /* ... */ }
int main () {
  for (int i = 0; i < 100; i++)
    if (max_speed > 250)
        expensive();
    else cheap();
}
```

Figure 2 A simple example.

```
Computing the balance information for the main function
Estimated cost of the function: 70804
1 accessible conditional statements
Delta 65000 at rex.c:22 in main (total count=100):
                       then=704; else=54;  // max_speed > 250
```

Figure 3 Analysis output: Δ-conditions for the example.

the configuration and/or the mode of the system, and by a *scenario* we denote a set of constraints on these parameters.

As an example, consider Figure 2 and suppose that `max_speed` is a variable among many parameters. Thus, the unbalanced conditional depends on variable values (parameters, configuration) that are not specified in the analyzed code. A system-expert knows that in a particular mode of execution or configuration (critical mode, model of vehicle, calibration) `max_speed` is less than 250. Restricting possible executions of the program by providing information about `max_speed` can therefore improve the WCET estimate for the configuration. Nevertheless, such a parameter first needs to be identified as relevant.

Figure 3 lists the Δ-conditions for the example, with a weight of 704 for the then-branch (Δ-weight of `expensive`) and a weight of 54 for the else-branch (Δ-weight of `cheap`). The conditional branches are executed within a loop, thus their weight needs to be scaled accordingly, resulting in a Δ-value of 65,000 over all loop iterations. As there is only a single `if`-statement present in the source the number of Δ-conditions is 1. The high Δ-value indicates that supplying additional annotations about variables involved in the condition have a high impact on the WCET estimate computed for the function. Therefore, our approach proposes this variable as a relevant parameter.

Supplying input annotations for those parameters helps determine the program paths that are valid in the current scenario. For example, assuming a system where `max_speed` is known to be < 200 reduces the WCET estimate from 250,033 to 29,933 cycles.

3 The Need for Concise Scenario Specifications

WCET analysis in an industrial context is applied with different aims in a number of development phases. A regular WCET estimate is often not enough, especially when the lack of context makes it highly imprecise. We overview these phases to illustrate where the precision of WCET analysis can be improved by incorporating system-expert supplied scenarios.

First, the WCET of a piece of code (e.g. a module or a function) is estimated in isolation of any influence from other modules, be it effects from input or output channels, relations with other modules, variable interdependencies, interrupts or the system configuration. These estimations happen soon in the development cycle, by the function developer. Second, in a module aggregation phase the focus shifts to the aggregation of a number of modules (10 to 40) to form a package suited to be reused in application projects. In this phase, the code is an assembly of modules and sequences and no more modified.

The tuning of these settings results in exclusive behaviours and can be expressed as limitations on valid program paths. Therefore, in this phase, information about top level system integration and other high level system information can be supplied to a WCET analyzer in order to gain a more realistic view on the WCET on the package level. Finally, an application project view point on the integration phase that combines a number of, e.g. 70 to 100, aggregates leads to a WCET estimate used for scheduling analysis, as well for a proper configuration of the task set. At this point, the top level system configuration is set-up, and execution modes can be defined (including assumptions about engine speed, coolant temperature, etc.). Specifying such assumptions about a mode in a scenario allows to infer a tight contextual WCET estimate for the particular mode and configuration.

In a complete automotive system there are up to 40,000 variables, some of them tightly coupled due to system state inter-dependency and closed loop effect. Scenarios are necessary in order to model influences on the system. The tool we propose helps the system-expert identify parameters that are likely to impact the WCET estimate and should thus be specified in the scenario. This way, the WCET analysis of a module during the aggregation phase, or of an aggregate during the project integration phase can be provide a value closer to the WCET in real setting, while only specifying a low number of parameters. Scenarios therefore include information on parameters such as:

- input and internal variables, corresponding to system states, values of acquisitions, information coming from the network or diagnosis data. For example, the set of active functions is influenced by the state of the engine system (full load, idle speed).
- configuration parameters, influencing arrays sizes, loop bounds, and (de)activate code branches in runnables, runnables, or even full aggregates. They can be
  - configured statically, and must be assumed during validation. Those are usually concerned with “high-level” configuration such as the number of cylinders, cylinder banks, the type of sensors or the type of combustion.
  - configured dynamically (calibration parameters) that can be modified later by tuning engineers and influence computations, like e.g. interpolation- and index-tables.

4 Branching Statements Analysis

The list of conditional statements shown in Figure 3 is computed by our *branching statement analysis*. For each conditional statements of a C program the analysis computes a Δ-value, i.e. an indicator of how unbalanced its branches are in terms of weights, (currently) a source based heuristic. Thus, the analysis requires no binary program nor architecture information. We implemented this analysis on top of the control flow analyser `oRange`.

Analysis Input. The analysis takes as input a C program, consisting of C source and header files, together with an entry point and optional input-annotations. User-supplied annotations may contain information about the program data and the program control flow, that might not be inferable from the program code.

Analysis Output. The analysis outputs a *list of branching statements* of the program. Each branching statement is accompanied by:

- information to localize it in the source code;
- an upper bound on the number of executions, N;
- a list of its valid branches, together with their branch-weights, w_i;
- its Δ-value $= N \times \max_{i,j}(w_i - w_j)$; the formula reduces to $N \times |w_{\text{then}} - w_{\text{else}}|$, respectively $N \times w_{\text{then}}$, for `if-then-else`, respectively `if-then`, statements.

The list is sorted by Δ-values in decreasing order and outputs the weight of the program.

Control Flow Pre-Analysis. We assume that the branching statements analysis is preceded by a control flow analysis that computes the following information:

- Loop bounds for each loop of the program
- Branch validity for each branching statement (`if`, `switch`)

This information is deduced from the code and/or supplied in the form of input-annotations and is used in the weight computation.

Abstract Syntax Trees. The C program is represented as a collection of functions, each defined by a name and an *Abstract Syntax Tree* (AST). An AST is composed of statements, structuring sub-statements and expressions, and expressions composing sub-expressions and function calls. Figure 4 depicts an example of an AST.

In absence of recursion, linking the function names to their corresponding AST yields a directed acyclic graph (DAG) which root is the entry point.

Branching Statement Analysis. Attaches the following information to each node:

- its weight, which is a pessimistic numerical indicator of the execution time of the function, statement or expression
- a set of conditional statements possibly evaluated during its execution, together with upper bounds on the number of times they are evaluated.

The information attached to a node is computed from the information carried by its children. In the example shown in Figure 4, assuming that the control flow analysis infers that both branches are valid, the weight of the IF statement is defined by (1) $w_e + \max(w_{\text{S1}}, w_{\text{S2}})$. The execution count of conditional statements in e, S1 or S2 are combined applying formula (1). An entry in the list of Δ-conditions is added for each conditional statement. Additionally, it lists the execution count and branches S1 and S2 together with their respective weight.

Information is computed in a bottom-up manner from (valid) leaves to the root. We rely on a heuristic assigning integer weights to nodes representing elementary program operations, e.g. numerical operations or memory accesses. Weights and counts are updated taking into account inferred or supplied loop bounds, while branches marked as invalid do not appear in the output. Branching statement with only one branch are not added to the list.

Finally, the root node carries the weight of the whole program and is output together with the set of Δ-conditions of the program.

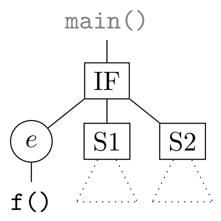

Figure 4 Example of Abstract Syntax Tree.

Limitations. The significance of the Δ-values depends on the constants and formulas used in the weight computation. Currently, and in our experiments, we use integer constants for different kinds of AST nodes. Nevertheless, a more complex scheme or using values computed by a static WCET analyzer is feasible. Statements which break the regular control flow do not receive a special treatment. As a consequence, a branch like `if (x) break;` inherently carries a low Δ-value.

5 Exploiting Scenario Specifications

An expert-supplied scenario, constructed by choosing and specifying values for suggested parameters, is likely to influence the computed WCET estimate. The control flow analyzer deduces path infeasibility, i.e. invalid execution under the current assumptions, from the additional annotations. Effects propagate to the low-level analyzer, as illustrated by the following example (Figure 5), as well.

In the following, we apply a simple cost heuristic for terminal nodes (cost 0 for numeric constants and field accesses and 1 for all other nodes) to identify scenarios and then use `oRange` and `OTAWA` to infer execution time estimates for the scenario.

Initially, the example is analyzed without scenario, finding all paths "valid". A follow-up low-level analysis computes possible addresses for the pointer access, resulting in $\top$ (no information) after the analysis of the `if`-statement. The assignment to `*p` results in a cache miss assumed by the low-level analysis.

Supplying value information (Figure 6) infers the branch invalid in the scenario and therefore marks the corresponding edge as `not-executed`. The low-level analyzer can thus ignore one of the branch for its analyses and infers an address for the access of `*p`, decreasing the original estimate, 95 cycles, to 83 cycles.

Before reporting on our experiments, let us summarize possible effects of utilizing Δ-conditions to specify concise but relevant scenarios:

The WCET path and estimate change when eliminating a branch on the WCET path. If a light branch was eliminated, the change is due to increased analysis precision of the following analyses. For example, when improved cache analysis results allow to infer block execution times (for previously selected blocks) that are below the execution time of alternative blocks (that were previously not selected).

The path is unchanged but the estimate is improved when a light branch is eliminated and the improvement of later analyses does not reduce block execution times of selected blocks below the execution time of their alternative blocks.

The WCET path changes while the estimate does not improve, in cases when a branch is eliminated but there exists another execution exhibiting a similar WCET estimate.

```
#include "missing.h"
int main ()
{
  int a, b;
  int *p;
  if (ext > 0)
    p = &a;
  else
    p = &b;
  *p = i;
}
```

Figure 5 Cache analysis fails to infer the address of `*p`.

```
// missing.h:
// no scenario, no value value
    ↪information
int ext;
(a)

// missing.h:
// scenario, additional value
    ↪information
int ext = 0;
(b)
```

Figure 6 "Invalidating" a path under a scenario arrows to infer the address of `*p`.

Finally, both the WCET path and the estimate are unchanged when eliminating a light branch and the improvement of following analyses does not propagate to blocks on the WCET path.

6 Experiments

Industrial Use-Case. The use-case is a 700 line C module provided by Continental Automotive France SAS. A system-expert manually identified and provided a list of 85 parameters and a scenario specification consisting of 30 parameter initializations. Branching statement analysis on the module reports 54 Δ-conditions, with Δ-values from 0 to 2034.

20 of the 30 parameters initialized in the provided scenario appear in the list of Δ-conditions, 18 of them exhibiting the highest 10 Δ-values of the list. 19 of the 54 Δ-conditions have low Δ-values (218 and less than 11) and no correspondence to the parameters in the scenario. As we rely on the parameter names to appear as operands in the Δ-conditions, a parameter may be linked to several Δ-conditions and vice versa.

Table 1 shows the result of WCET analysis of the module: column 1 lists the provided scenario, column 2 lists the number of specified parameters in the scenario and column 3 to 6 list the WCET estimate and improvement compared to the global WCET for an ARM7 lpc2138 platform, without and with a 1KB direct mapped data cache.

WCET analysis of the module without scenario, (1) global, reports 2553 (6883) as WCET estimate. WCET analysis of the expert-provided scenario, specifying 30 parameters, (2) full scenario, yields an improvement of 5%. Rows, (3)-(6), list the estimate and gain when specifying only those parameters involved in the i highest valued Δ-conditions.

To investigate the correspondence between high Δ-values and gain in the WCET estimate we inverted the scenario that initializes 3 parameters, row (3), thus forcing execution of the light branches of the corresponding conditionals, in row (7). To validate that specifications for parameters not contained in the list of Δ-conditions have little impact on the estimate, we supply the 10 parameter initializations that do not appear in any Δ-conditions, row (8).

Summarizing, branching statement analysis identified 20 of 80 parameters as important due to their high Δ-values in the list and they coincide with specified values in the expert-provided scenario. 10 parameters specified in the expert-provided scenario do not appear in the Δ-condition list and have almost no impact on the WCET estimate, while specifying only parameters identified in the 10 highest Δ-conditions still improves the estimate.

The experiment shows that our branching statement analysis can help system-experts focus on the relevant parameters from the vast number of possible parameters.

Table 1 WCET computation depending on parameters provided in scenarios

scenario	# parameters	no cache		cache	
(1) global, no scenario	0	2553	gain	6883	gain
(2) full scenario	30	2426	5%	6486	5.7%
(3) 3 highest Δ	3	2553	0%	6833	0%
(4) 8 highest Δ	10	2479	3%	6679	3%
(5) 9 highest Δ	14	2463	3.5%	6623	3.8%
(6) 10 highest Δ	18	2448	4%	6568	4.6%
(7) inverted 3 highest Δ	3 (inverted)	2055	19%	5795	15.8%
(8) none of Δ	10	2551	0.08%	6831	0.03%

Table 2 Potential for WCET improvement for Mälardalen benchmarks.

program	**# Δ**	**highest Δ**	**overall weight**	**ratio**
sqrt	4	4458	4540	98.19
expint	3	35800	38026	94.14
prime	4	12,773,225	86,858,003	14.70
crc	5	6144	49223	10.45
ndes	5	384	43930	0.8
st	4	743	114,137	0.65
fir	2	30	204,371,956	0.00001

Branching Statements Analysis of Mälardalen. Even though our approach is motivated by industrial need, branching statement analysis provides relevant results when applied to the Mälardalen benchmark suite [6]. The benchmarks lack scenarios, therefore we target identifying interesting variables and program points instead of system parameters.

Column # Δ in Table 2 states the number of unbalanced conditionals found in the program. Highest Δ lists the highest Δ-value reported. The last two columns list the weight of the program and its ratio to the highest Δ-value. The ratio is an indication of potential improvement when supplying additional information for the conditional is feasible.

In programs like `expint` or `sqrt`, the highest Δ-value weighs over 94% of the total weight, which hints at a relevant program point[1]. There is potential for improvement in programs `crc` and `prime`, yet they lack the opportunity to specify annotations. The low Δ-values of the rest of the programs suggest a low potential for improvements.

As expected, the nature of the benchmarks, lacking system parameters, prohibits scenario specification, but interesting program points can still be identified by the analysis.

7 Related Work

There is a body of work that is related to branching statement analysis.

The weight computation and propagation in the AST is comparable to tree-based WCET computation, whereas identifying relevant conditionals is related to (static) profiling. In contrast to the other approaches, it aims at helping system-experts identify relevant parameters

[1] In `expint` the high weight of the conditional is due to an inner loop with a high execution count, while it is run only once. In `sqrt` the condition guards a light special case of the computation.

by identifying unbalanced conditionals on source level using a simple heuristic for weights.

Tree-based WCET computation can be applied as alternative to IPET [3]. Estimates are computed by a bottom-up traversal of the parse tree of a program. Leaves represent basic blocks and are annotated with timing information. The program is traversed and for each node timing information is computed from the timing information of its child nodes. In contrast to tree-based WCET computation, branching statement analysis does not rely on timing information of basic blocks but uses a heuristic to compute timing information from the syntactic expressions.

In [2] the authors present the *criticality metric* to express for each statement how important it is for the global WCET. This allows finding out, for a piece of code, how close the WCET of paths passing through the piece of code is to the global WCET. The ratio between Δ-values and the weight of the function indicates a potential for improving the WCET estimate by supplying a scenario.

Dynamic program profiling usually executes an instrumented program in order to explore the performance and/or flow [5]. Static approaches generate static profiles, e.g. by computing probabilities for decisions at branching points [9]. A major problem in profiling is to find input values that capture the performance profile of the application. Our analysis does not execute or instrument the program and helps to identify relevant parameters.

The author of [8] sketches the ingredients for a static profiler. The output is an unfolded inter-procedural control flow graph computed from the results of multiple static analyzers that identify feasible paths and compute loop bounds. The graph is guaranteed to include the worst-case behaviour of the program. Static analysis results are used similarly in branching statement analysis but instead of execution frequencies it computes weights.

Most closely related is a work on scenario detection [4], following a comparable approach, with a slightly different goal. The influence coefficient is computed for parameter variables, which are identified as variables or fields that are assigned once and do not change over the program execution. Scenarios are constructed such that they split and cover the domain of the parameters, allowing to WCET analyze each scenario. The maximum execution time among the scenarios is then considered as the WCET of the application. In contrast to [4], our notion of parameters allows for changes in value, and they might not be assigned in the code at all. Instead of discovering variables, we match conditions with high Δ-values to a number of pre-selected parameters. Furthermore, our weight heuristic should be independent of both the underlying architecture and WCET analysis, in order to be able to apply it in different phases of development.

8 Conclusion and Outlook

We presented branching statement analysis, an approach to guide system-experts in generating concise, but relevant, scenarios for the WCET analysis of systems. Such systems are usually composed of a number of components, influenced and controlled by a vast number of parameters. The high number of parameters usually results in an overly pessimistic WCET estimate and prohibits exhaustive specification of scenarios that allow to obtain a more realistic estimate. Branching statement analysis allows to find unbalanced conditions that depend on parameters. Additional information about a low number of identified parameters can already significantly improve the WCET estimate or validate the estimate for the scenario. To this end, we see branching statement analysis as an important step towards a more functionally representative (or plausible) WCET, instead of a purely structural one.

We are currently applying part of our approach manually, i.e. we rely on scenarios specified by system-experts. To further improve trust, such scenarios could be verified, whenever possible, by automated tools.

We plan to integrate our approach in a fully automatic setting, where additional information is inferred using static analysis. Branching statement analysis could be used to select program points where counter instrumentation and analysis is applied to infer constraints between instrumented basic blocks.

Additional effort will be put into a richer set of supported input-annotations in the scenario, currently restricted to information about variable values and boolean information about the execution of conditional branches. The system-expert might have at its disposal information like limits on the number of execution of a statement or contexts in which a statement must or must not be executed.

References

1 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *Proc. of IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

2 Florian Brandner, Stefan Hepp, and Alexander Jordan. Static Profiling of the Worst-case in Real-time Programs. In *Proc. of RTNS*, pages 101–110, 2012.

3 Matthew Emerson, Sandeep Neema, and Janos Sztipanovits. *Handbook of Real-Time and Embedded Systems*, chapter 6. CRC Press, 2006. ISBN: 1584886781.

4 Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten, and Henk Corporaal. Automatic Scenario Detection for Improved WCET Estimation. In *Proc. of DAC*, pages 101–104, 2005.

5 Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proc. of SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

6 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. of WCET*, pages 136–146, 2010.

7 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 *Proc. of RTCSA*, Taiwan, 2008.

8 Adrian Prantl. Towards a Static Profiler. Technical report, Vienna University of Technology, 2009.

9 Youfeng Wu and James R. Larus. Static Branch Frequency and Program Profile Analysis. In *Proc. of MICRO*, pages 1–11, 1994.

10 Jakob Zwirchmayr, Armelle Bonenfant, Marianne de Michiel, Hugues Cassé, Laura Kovacs, and Jens Knoop. FFX: A Portable WCET Annotation Language. In *Proc. of RTNS*, pages 91–100, 2012.