The WCET Analysis using Counters - A Preliminary Assessment

Remy Boutonnet
Verimag, UJF
Grenoble, France
remy.boutonnet@imag.fr

Mihail Asavoae
Verimag, UJF
Grenoble, France
mihail.asavoae@imag.fr

ABSTRACT

The Worst-Case Execution Time (WCET) analysis aims to statically and safely bound the longest execution path of a program. It is also desirable for the computed bound to be as close as possible to the actual WCET, thus making the result of a WCET analysis tight. In this paper we propose a methodology for the WCET analysis using an abstract program (with special program variables called counters), in order to better exploit the underlying program semantics (via abstract interpretation) and to produce potentially tighter WCET bounds.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

WCET Analysis, abstract interpretation, semantic analysis

1. INTRODUCTION

The interaction of hard real-time systems with their external environment is governed by a set of timing constraints. In order to solve these constraints, it is necessary to estimate the worst-case execution time (WCET) of the system components. A WCET analysis computes for a given task or program, a WCET bound which should be safe and tight (close to the actual WCET).

The WCET analysis is performed at the binary level, with knowledge about the underlying architecture. First, the typical WCET analysis workflow as standardized in [18], extracts the control-flow graph (CFG) from the binary code. Subsequently, this CFG is the working structure for both flow- and architecture-related analyses. Their result, which is an annotated CFG, is used to compute the WCET bound in a final phase, called path analysis. In order to achieve tight bounds, a WCET analysis relies on a number of specific analyses, from flow analysis (e.g. detection of loop bounds and infeasible paths) to architecture analysis (e.g. of caches).

In this work, we address the flow analysis from the following angle: how to extract more accurate semantic properties which can help the WCET analysis by removing some infeasible paths. We denote an analysis which extracts semantic properties, invariants on program executions, as a *semantic analysis*. More specifically, in our context a semantic analysis targets invariants which are directly translated into integer linear programming (ILP) constraints, as emphasised by the implicit path enumeration technique (IPET) [17]. Our work uses a program analyzer called Pagai [11], at the LLVM Intermediate Representation level, over the LLVM compiler infrastructure [15]. The key element of our approach is the extraction of invariants from an abstract representation of the input program, as an instrumented code with *counters*. A counter is a special program variable which is attached to a program part (e.g. a basic block) and incremented every time the control flows through that part. We propose the following workflow: after an automated instrumentation of LLVM-IR code, the semantic analysis is performed using Pagai (with a linear arithmetic abstract domain [6]). The invariants, as relations between counters, are transferred to binary code (actually to the path analysis formulation of Otawa [4]), using a block-level traceability tool. We measure the improvements on the WCET bounds on a set of syntactic and standard benchmarks.

```
x = 0; i = 0; s = 0;          // α = 0; β = 0; γ = 0;
while (i < N) {               // α++;
    if(x < 10){
        s += 3;               // β++;
    }
    if(s < N){
        s += 2; x ++;         // γ++;
    }
    i ++;
}
```

Figure 1: Our example program instrumented with counters α, β, γ

We use the program described in Figure 1 to advocate on how the WCET analysis can benefit from a counter-based approach. The counters are α, β, γ, and N is a loop bound.

The following relations can be derived on those counters, which are satisfied at the end of the program:

$$\beta + \gamma \leq \alpha + 10 \quad (1)$$

It shows that the blocks 4 and 6 of the control flow graph of this program, in Figure 2, are both executed in the same iteration of the while loop at most 10 times. Therefore, this information is interesting for us since it leads to refinements in the WCET of the whole program.

Outline. In Section 2 we overview the existing methods on extraction of semantic properties for the WCET analysis.

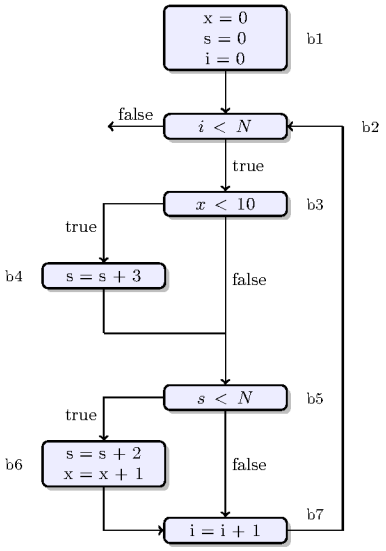

Figure 2: The control flow graph corresponding to this example.

In Section 3 we elaborate on how our system is designed, implemented and experimented with. We draw conclusions and discuss about directions of future work in Section 4.

2. RELATED WORK

The WCET analysis requires knowledge about loop bounds and infeasible paths in order to compute tight WCET bounds. The existing support for automatic extraction of semantic properties like the aforementioned ones is usually coming in two flavours: as a result of abstract interpretation or of symbolic execution. More recent approaches also use state of the art satisfiability modulo theory (SMT) solving to address parts of the WCET analysis.

Abstract interpretation, introduced in [5] is one of the major program reasoning techniques. It relies on abstract domains, like linear arithmetic [6] and fixpoint computation to generate invariants at the program points of interest. In the context of WCET analysis, abstract interpretation plays a key role [19] in both control-flow (e.g. in value analysis) or processor behaviour analyses (e.g. in cache analyses).

Symbolic execution [13] is a technique which uses arbitrary values as program inputs and allows program reasoning at the level of execution paths. Symbolic execution has drawn interest from the WCET analysis community [9, 14], but while it is potentially very precise, it suffers from scalability issues when it is applied on large programs. As a consequence, symbolic execution in WCET analysis is coupled with search space reduction [9] techniques or it is applied on code fragments [14].

Recent works [14, 10] rely on SMT-solving techniques to compute WCET bounds. For example, WCET squeezing [14] employs a form of symbolic execution on paths returned by an external WCET analyzer. This technique embeds the path analysis into a CEGAR loop, allowing an incremental strengthening of the WCET bound. The approach in [10] computes WCET bounds as solutions of optimisation modulo theory problems (i.e. extensions of the SMT to maximisation problem). The program semantics are encoded as an SMT formula. To maintain the scalability of the analysis, the original SMT formula is augmented with additional constraints called "cuts", which express summaries of portions of code.

Using counters to extract semantic properties is not new, existing counter-based approaches have been proposed, for example in [8] using a single counter and in [12] with multiple counters. The former considers one counter which represents time and accumulates the program semantics into it. The later overcome the issue with the single-counter annotations to work with complex invariant generation tools, CFG graph transformation and generation of progress invariants, it computes loop bounds and infeasiblilty relations.

The existing WCET analyzers span from industrial strength platforms, like aiT [1] to academic tools like Otawa [4], SWEET [2] and Chronos [16]. The oRange tool [7] complements the Otawa timing analyzer by computing loop bounds using static analysis with abstract interpretation [5, 3] on C programs. The SWEET tool supports an implementation of the abstract execution over the domain of intervals. The Chronos timing analyzer integrates a pattern-based semantic analysis which keeps track, for a particular branch, of which assignments or other branches may influence it. The industrial timing analyzer aiT uses a similar pattern-driven analysis to identify code portions (e.g. loop or branch shapes) and apply the appropriate analysis. While we conducted limited experiments on SWEET, Chronos and aiT, these tools seem capable to detect certain types of bounds and infeasibility relations: for SWEET - up to the strengths of the abstract domain and for Chronos and aiT - up to the code structure which exhibits certain syntactic patterns. However, using a specialised static analyzer to compute semantic properties and to transfer these properties seems to offer power (through various abstract domains) as well as flexibility (i.e. driven by the strengths of the static analyzer).

3. SYSTEM DESIGN AND IMPLEMENTATION

3.1 General System

In the most general context, a counter-based methodology for WCET analysis is driven by two elements: a compilation toolchain (which also fixes the input language) and a WCET analyzer (which includes the necessary processor behaviour analyses). Next in Figure 3 we describe an implementation of this methodology over the LLVM infrastructure and the Otawa timing analyzer.

The standard WCET analysis workflow, in Figure 3 (left) computes the WCET bound of the binary code (in our case it is ARM code) generated from LLVM-IR code. The Otawa timing analyzer relies on an ILP formulation of the path analysis and embeds an ILP solver to compute the result (i.e. the WCET bound). Our counter-based analysis workflow, in Figure 3 (right) could be seen as a plugin for the WCET analysis. For the purpose of semantics extraction, the Pagai static analyzer uses the initial LLVM-IR code, instrumented with counters. Pagai computes invariants on the counters, at the LLVM-IR level, using either abstract interpretation or its combination with SMT solving. The invariants are directly translated into ILP constraints. Fi-

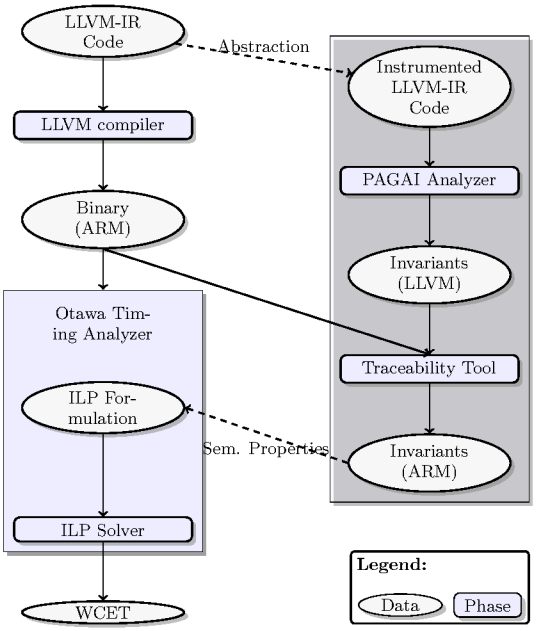

Figure 3: General workflow for Counter-based WCET Analysis

nally, a traceability tool maps the LLVM-IR blocks to ARM basic blocks and facilitates the transfer of Pagai invariants on counters at the binary level. These properties are integrated into the path analysis of Otawa and solved in order to obtain the WCET bound. The WCET analysis is performed over the initial program, the instrumented code is used only to extract invariants w.r.t. program semantics. Moreover, our workflow does not consider code optimisations between the LLVM-IR and ARM levels, however the code could and should be optimized from C (i.e. not represented in the workflow) and LLVM-IR levels. Our counter-based methodology works when replacing Otawa with the aiT, SWEET or Chronos timing analyzers (all using IPET), up to some supported architectures. The counters approach can be used in difference scenarios: to find or refine loop bounds or infeasible paths, particularly those created by mutually exclusive conditions.

By processing the example of Figure 1 through the PAGAI static analyzer, we automatically obtain the following constraint between the counters α, β and γ:

$$10 + \alpha - \beta - \gamma \quad \geq \quad 0 \tag{2}$$

as well as $-\alpha + 3\beta + 2\gamma \geq 0$, $\alpha - \beta \geq 0$ and $\alpha - \gamma \geq 0$.

This relation shows what we have derived by hand for our example: the then-parts of the two conditions in the while loop are both executed in the same iteration at most ten times. Therefore, the counters approach with a static analyzer is able to automatically find a non-trivial case of infeasible path: the path containing the blocks b4, b5, b6, b7 is executed at most ten times.

We can also use the counters approach to find or refine a loop bound where other tools like oRange cannot find one. In our example in Figure 1, if the condition of the while loop $i < N$ is replaced by $x < 10$ and the second condition $s < N$ by $s < 150$, the PAGAI static analyzer outputs the relation $-10 + \alpha = 10$ which enables us to show that the while loop is executed at most 10 times in that case.

3.2 Experiments

Our set of benchmark programs covers a wide range of applications (though of small size - column **LOC** in Figure 4). We include automatically generated code from high-level designs - (e.g. a snapshot called `selector` and avionics-specific controllers in `roll-control` and `cruise-control`) as well as several syntactic programs with complicated infeasible paths (e.g. `sou`, `even`, `break`, and `rate_limiter`). In order to extract semantic properties using the Pagai static analyzer, we automatically instrument the LLVM IR code (the working level for Pagai) with a number of counter variables (in column **#Cntrs**) and a set of invariants (in column **#Inv**) which are fed into the ILP representation of path analysis. In this paper, we use the processor behavior analysis as provided by Otawa, our main concern being the program-level semantic analysis.

The set of invariants covers relations between basic blocks (represented by their respective counter variable) of several forms. First, there are the loop bounds types of relations, like for the benchmarks `break` and `selector`. Second, the path infeasibility relations are expressed either as invariants on two counter variables (in the case of pairwise exclusive branches) - for the benchmark programs `sou` and `even` - or as a counter value which is equal to zero (i.e. the paths going through the particular basic block are infeasible) - for the benchmark `rate_limiter`. Moreover, for some benchmarks like `selector`, the infeasibility relations are more expressive as an invariant on three counter variables than the pairwise relations. Third, the set of invariants also includes relations which do not contribute to a reduction of the WCET bound. Overall, the experiments show promising results because, in general it is difficult to obtain improvements of more than several percents (indeed, the code size is rather small).

4. CONCLUSIONS

In this paper we addressed the problem on how to tune the WCET analysis so as to produce tighter WCET bounds. As such, we proposed a methodology to extract semantic information via special program variables called counters. We used a program analyzer, called Pagai, to compute flow relations (as relations between these counters). Finally, we transferred these relations into an IPET formulation of the path analysis and observed encouraging results with improvements over 20% on certain benchmarks. The methodology is still under development as we would like to investigate how our counter-based WCET analysis compares with and could complement existing WCET analyzers using automated extraction of semantic properties.

5. ACKNOWLEDGMENTS

The authors thank Fabienne Carrier and Claire Maiza for their valuable comments on the paper content and Julien Henry for his help with the Pagai static analyzer. This work was partially funded by grant W-SEPT (ANR-12-INSE-0001) from the French *Agence national de la recherche*.

6. REFERENCES

[1] AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers.

Name	Program Description	LOC	#Cntrs	#Inv	WCET init	WCET final	Red %
`selector`	Fragment of SCADE design	134	14	14	1112	528	52.6%
`roll − control`	From the SCADE Suite	234	25	19	501	501	0%
`cruise − control`	From the SCADE Suite	234	35	31	881	852	3.3%
`sou`	Syntactic benchmark 1	69	3	3	99	67	47.8%
`even`	Syntactic benchmark 2	82	9	8	2807	2210	21.3%
`break`	Syntactic benchmark 3	114	4	5	820	820	0%
`rate_limiter`	Program from [10]	35	2	2	43	29	32.6%

Figure 4: Set of benchmarks for the counter-based WCET analysis

[2] http://www.mrtc.mdh.se/projects/wcet/sweet/index.html.

[3] Z. Ammarguellat and W. L. H. III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *PLDI*, pages 283–295, 1990.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *SEUS*, 2010.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[7] M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *RTCSA*, 2008.

[8] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[9] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, 2006.

[10] J. Henry, M. Asavoae, D. Monniaux, and C. Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *LCTES*, pages 43–52, 2014.

[11] J. Henry, D. Monniaux, and M. Moy. Pagai: A path sensitive static analyser. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

[12] N. Holsti. Computing time as a program variable: a way around infeasible paths. In *WCET*, 2008.

[13] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[14] J. Knoop, L. Kovács, and J. Zwirchmayr. Wcet squeezing: on-demand feasibility refinement for proven precise wcet-bounds. In *RTNS*, pages 161–170, 2013.

[15] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[16] X. Li, L. Yun, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007.

[17] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[18] R. Wilhelm and all. The worst-case execution-time problem—overview of methods and survey of tools. *ACM TECS*, 7(3):1–53, 2008.

[19] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. In *CAV*, pages 22–36, 2008.