Structured Testing of Worst-Case Execution Time Analysis Methods

Jakob Engblom[†*]
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
email: jakob.engblom@iar.se

Andreas Ermedahl[†]
Dept. of Information Technology
Uppsala University
Box 325, SE-751 05 Uppsala
Sweden
email: andreas.ermedahl@docs.uu.se

Friedhelm Stappert[‡]
C-LAB
Fürstenallee 11
33102 Paderborn
Germany
email: fst@c-lab.de

Abstract

Knowing the Worst-Case Execution Time (WCET) of a program is necessary when designing and verifying real-time systems. When WCET analysis tools are used to estimate the WCET, the tool is a critical part of the system design and must be correct.

In this paper we present a methodology for systematically testing WCET analysis tools. The methodology is based on a decomposition of WCET analysis into a set of components that should be tested and validated in isolation. Our testing methodology does not require that we have a perfect model of the hardware, and the validation of the hardware model is considered as a separate problem. We illustrate the usage of our testing methodology for the pipeline analysis and calculation phase of our WCET analysis method.

Keywords: WCET, pipeline analysis, hard real-time, embedded systems, testing, validation.

1. Introduction

The purpose of *Worst-Case Execution Time* (WCET) analysis is to provide a-priori information about the worst possible execution time of a program before using the program in a system.

WCET estimates are used in real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times.

To be valid, WCET estimates must be *safe*, i.e. guaranteed not to underestimate the execution time. To be useful, they must be *tight*, i.e. provide low overestimations. The goal is to come as close as possible, but not below, the execution time on the target hardware.

The WCET depends both on the program flow (like loop iterations and function calls), and on architectural factors like caches and pipelines. Thus, both the program flow and the hardware the program runs on must be modelled in a WCET analysis.

When evaluating WCET analysis methods, the common methodology is to compare a WCET estimate with an execution of the same program with known worst-case data on the target hardware.

This evaluation method is problematic, since it mixes the effects of several sources of errors. For example, errors in program flow analysis and the hardware model used by the method, and the method itself, may cancel each other. Also, if errors are detected, it is very hard to pinpoint the error source.

In this paper we present a testing methodology designed to isolate the potential errors in each component of a WCET analysis. We demonstrate the applicability of the methodology on the pipeline analysis and the calculation phase of our previously published WCET analysis method [1].

2. WCET Phases and Related Work

To generate a WCET estimate, we consider a program to be processed through the phases of *program flow analysis*, *global low-level analysis*, *local low-level analysis* and *calculation*.

The program flow analysis phase determines the dynamic behaviour of the program. The flow analysis will provide information about which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc.

---

*Jakob is an industrial PhD student at IAR Systems (http://www.iar.com) and Uppsala university, sharing his time between research and development work.

†This work is performed within the Advanced Software Technology (ASTEC, http://www.docs.uu.se/astec) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, http://www.nutek.se).

‡Friedhelm is a PhD student at C-LAB (www.c-lab.de), which is a cooperation of the University of Paderborn and Siemens AG.

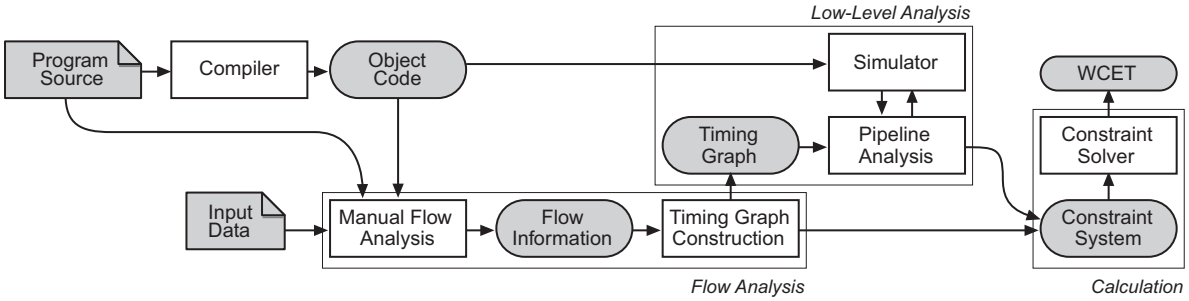

Figure 1. Overview of our current WCET analysis system

The purpose of the low-level analysis phases is to extract the times for an instruction or sequences of instructions taking into account timing effects of different hardware features.

The global low-level analysis determines the effects of caches, branch predictors, and other machine-level effects that must be analyzed across the entire program. Example results are safe estimates on cache hits and misses for instructions and data.

The local low-level analysis determines the effect of the target machine pipeline, memory configuration, bus speeds, etc. These are machine timing effects that can be handled locally, for single instructions or basic blocks and their immediate neighbors.

The calculation phase calculates the WCET estimate for a program, given the program flow and global and local low-level analysis timing results. The result is the path or execution profile that corresponds to the execution leading up to the maximum execution time of the program.

3. Validating WCET Tools

To guarantee that a WCET estimate produced by a WCET analysis tool is safe and tight we must guarantee that each analysis phase is safe and tight in its own right. Otherwise, errors in one phase could mask errors in other phases. Also, we must make sure that the integration of results of different analysis phases does not generate new errors.

Errors may be due to implementation bugs, incorrect analysis methods, and false target platform assumptions.

Testing is the only viable method for checking the correctness of WCET tools, since even if a method has been proven correct, its implementation may still be in error. Also, testing is the only way to compare to the target hardware.

Testing each phase in isolation is not trivial, since each phase builds on the output of the previous phase(s). The solution is to make sure that the phases preceeding the phase you want to test are *frozen*. By this we mean that the same set of information from the phase is used both in the method being tested and the reference (the independent, correct, result we compare to).

In the following sections we will present our WCET analysis tool and how we apply our testing methodology of component isolation to our tool in order to test the correctness of the local low-level (pipeline) analysis and the calculation method.

3.1. Architecture of our WCET Tool

Figure 1 gives an overview of our WCET analysis system as implemented today. It is a concrete implementation based on the principles presented in [1].

Since the architecture is very modular, testing is simplified. Each component has clearly defined roles and interfaces, and to produce a correct tool, we want to test the correctness of each component in isolation.

We manually inspect the object code, source code, and input data of the test programs, and construct a description of the worst-case program flow using the flow information description language presented in [2]. Automatic flow analysis is considered for the future.

We do not include any cache or other global low-level effect analysis in the current version of the tool, since our target system does not have a cache (the current target is the NEC V850E CPU, a typical embedded pipelined RISC microcontroller).

Our local low-level analysis are based on running basic blocks and sequences of basic blocks through a *Simulator* extracting the execution times of basic blocks and the timing effects of pipeline overlap between blocks. The simulator is assumed to be trace-driven, which means that it does not have a semantic model of the CPU. It only models the pipeline behavior, given a stream of instructions.

We use an IPET-style (Implicit Path Enumeration Technique) calculation [3, 4, 7] where program flow and atomic execution times are represented using algebraic and/or logical constraints. The WCET estimate is calculated by maximizing an objective function, while satisfying all constraints. At present, we use the constraint solver of a Prolog system, but are working on

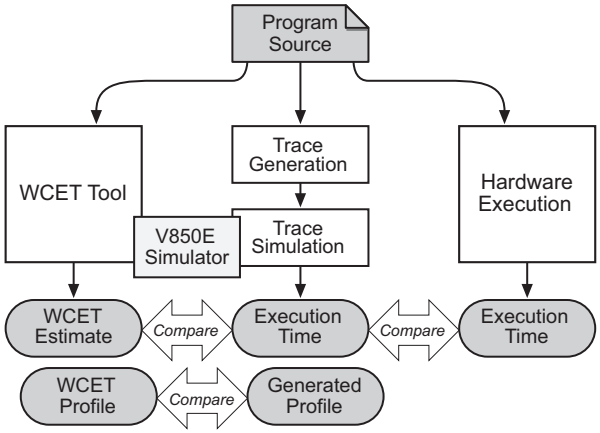

Figure 2. Overview of our testing method

writing our own special-purpose solver to increase the efficiency.

Figure 2 gives an overview of our testing method. The comparison between the output of the WCET tool and the trace simulation will be used to determine whether the algorithms of the WCET analysis method is correct (regardless of the hardware model), and the comparison to the real hardware will indicate the quality of the hardware model.

3.2. Obtaining a Known WCET

In order to check the correctness of our tool, we need to have a known worst-case execution to compare to. This is obtained by executing instrumented test programs with known worst-case data on a workstation, generating an *execution trace* showing how the programs execute. The trace is a long list of basic block names, corresponding to the order in which the blocks are executed.

The trace is used to drive the CPU simulator. The result is an execution time corresponding to an execution of the program on the simulator. Note that we do not run the program on the hardware, since this would add the potential errors in the simulator to the potential errors in the calculation method and pipeline analysis. Comparing the simulator to the hardware is a separate issue.

The trace is also used to obtain an *execution profile* for the worst-case execution. The profile is a count of how many times each basic block in the program is executed (with no sequence information). This information is compared with the program flow information resulting from the WCET analysis.

3.3. Freezing Other Components

The calculation method and pipeline analysis are isolated by making the other components of the WCET analysis method constant. To perform a complete WCET analysis, we need some results from other analysis phases, and thus we need to ensure that errors here do not affect the testing of the pipeline analysis and calculation.

In order to avoid errors due to differences in input data between different runs, the input data has to be fixed and produce a worst-case execution for all our experiments. This is achieved by manual code inspection and systematically doing testruns.

The program flow analysis is the most important potential source of errors. Our solution here is to use simple programs where the flow is easy to deduce and known (since we know the input data, that source of uncertainty is removed). We check the correctness of the program flow model and that it corresponds to the WCET execution by inspection.

Our target hardware does not have caches or branch predictors, thus, no global low-level effects will be present and no global low-level analysis is needed. If we had been using a cached processor, we would have disabled the caches to remove that source of variability.

3.4. Testing the Calculation Method

For the calculation method we want to show that the use of a constrained maximation problem to model program finds the longest executable path through the program.

We take advantage of the fact that the IPET maximization process generates a program execution profile: the resulting execution counts are mapped back to the corresponding basic blocks to get a profile that is comparable to that of the trace generation.

If the execution profiles from the WCET analysis and the trace generation agree, the IPET calculation has managed to find the correct flow for the worst-case execution time.

3.5. Testing the Pipeline Analysis

Errors in the pipeline analysis might be hidden by the calculation method, but if the execution profiles of the WCET tool and the worst-case execution agree, then any difference between the trace-driven execution time and the final execution time estimate from the WCET tool will be due to errors in the pipeline analysis.

3.6. Testing the Simulator

In order to get some indication of the overall quality of our approach, we compare our results to execution on the real hardware. The outcome of this is not considered a relevant test for the correctness of WCET analysis algorithms. Rather, we investigate the quality of the hardware model employed.

4. Experiments

We have used some simple benchmark programs to test our WCET tool. All the programs have been used by other groups [5, 6], and their worst-case behavior is known.

Since we are targeting embedded systems, we had to make slight modifications to the benchmark programs: no operating system calls are allowed, which means that input data for input-dependent programs was integrated into the program, and that calls to library functions (like `printf()`) were removed.

Program	Analysis	Sim	Real
`fibcall`	286	286	312
`matmult`	239528	239528	222236
`jfdctint`	5550	5550	4843
`insertsort`	1249	1249	1080
`duff`	1226	1226	1081

Figure 3. Measured execution times

We have run five test programs through our WCET tool, through trace generation and simulation, and on real hardware. The results of the timing measurements are shown in Figure 3. All times are measured in clock cycles.

Execution times on real hardware are obtained by running the programs on a V850E emulator. The emulator was set up to emulate a single-chip configuration with 60kB of internal RAM and 64kB of internal ROM. No I/O was used.

4.1. Results

The profiles generated by the WCET tool and the real execution match exactly (not shown in any table, since that would be a rather boring long list of numbers).

This indicates that the constrained maximization does find the worst-case path, and that it does not push the execution counts beyond the actual worst case. Thus, the IPET-based modeling and calculation are shown to be functioning (at least for our set of test programs).

Since the execution profiles match, the agreement in execution time between the WCET tool and the trace-driven simulation indicate that the pipeline analysis method is correct.

Compared to the hardware, the simulated execution times for four of the programs are between 5 and 15% greater. Thus, the simulator usually overestimates the execution time. Interestingly, for `fibcall` the simulated time is *less* than the hardware execution time.

This means that although the calculation and pipeline analysis are correct, the simulator still introduces errors that makes the complete system inappropriate for use in cases where safety is necessary. Thus, we need to improve our simulator to use the current tool for real WCET analysis work.

5. Conclusions and Future Work

In this paper, we have dealt with the issues involved in validating a WCET analysis method. For safety-critical systems development, it is necessary to validate WCET analysis methods.

We have demonstrated how WCET analysis can be decomposed into a set of components, each of which should be validated in isolation. Only by composing well-tested and safe components is it possible to build a reliable WCET tool.

We have presented a testing methodology for isolating the pipeline analysis and calculation method from the program flow analysis, cache analysis, and hardware modelling (CPU simulator). We have applied this methodology to our previously published WCET algorithm [1], and given evidence that the pipeline analysis and calculation method are safe and tight.

We intend to extend the methodology as more components and more hardware platformas are added to our WCET architecture. At the moment, we have a Master's Thesis project underway that aims to remove the errors in our V850E simulator by systematic comparison between the simulator and the hardware.

References

[1] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, December 1999.

[2] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000. Accepted for publication.

[3] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[4] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[5] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[6] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, May 2000.

[7] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.