Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation *

Raimund Kirner, Peter Puschner, Ingomar Wenzel
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
{raimund,peter,ingo}@vmars.tuwien.ac.at

Abstract

Traditional worst-case execution time (WCET) analysis methods based on static program analysis require a precise timing model of a target processor. The construction of such a timing model is expensive and time consuming.

In this paper we present a hybrid WCET analysis framework using runtime measurements together with static program analysis. The novel aspect of this framework is that it uses automatic generation of test data to derive the instruction timing of code sequences. Program paths are decomposed into subpaths to make execution-time analysis based on runtime measurements feasible.

1 Introduction

The current situation on WCET analysis is not satisfying, because widely used industrial-strength WCET analysis tools are still missing [7].

One challenge of WCET analysis is the variable instruction timing of processors. Complex processors have performance increasing features like caches or pipelines that maintain an internal state that depends on the execution history. Precisely modeling these features is problematic as on the one side it becomes quite complex and on the other side, exact information of the previous instruction stream cannot be calculated in general. A further problem is that the vendor's documentation of a processor's instruction timing is often a very rough approximation of reality. Problems arising on WCET analysis using static hardware modeling are described in [5].

A further problem of static WCET analysis is that it is very time consuming to model features of complex processors and furthermore, it has to be done for each processor for which WCET analysis is required. The alternative is to use measurement-based WCET analysis. However, simply performing exhaustive end-to-end measurements is not feasible for real-size programs. Therefore, measurement-based WCET analysis is used in combination with static analysis techniques. Approaches to hybrid WCET analysis do already exist [4, 1] but research in this area is just at the beginning. The path analysis problem of static WCET analysis is currently shifted to the problem of generating test data for measurement-based approaches. The current approaches require the user to provide test data or simply use random testing. In addition to runtime measurements, Ernst and Ye propose to switch back to traditional static WCET analysis techniques in case that the test data provided by the user did not cover all program blocks [4].

In this paper we present a measurement-based WCET analysis framework with automatic generation of test data. The problem of automatically generating the test data is tackled by standard program analysis techniques like *model checking* [2, 9] or *constraint-based analysis* [10]. The approach is based on decomposition of program paths into subpaths of program segments. A static WCET calculation method is used after the instruction timing of subpaths of program segments has been assessed by runtime measurements.

The paper is structured as follows: Section 2 gives a discussion about demands from industry for the use of WCET analysis tools. The measurement-based WCET

*This work has been supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)".

analysis framework is described in Section 3. Section 4 discusses technical aspects of the framework. Finally, Section 5 concludes this paper.

2 Requirements for an Industrial-Strength WCET Analysis Tool

Before describing our new WCET analysis method, we give a motivation for its development by describing the industrial needs on a WCET analysis method. Previously proposed WCET analysis methods often only demonstrate several analysis capabilities without showing their applicability in an industrial environment. To be more precise, the following list gives demands for a WCET analysis tool raised by people working in industry. This list also contains aspects regarding the use of modeling tools like MATLAB/Simulink, as they are increasingly used in industrial software development.

1. The tool must work with minimal user interaction. In particular, it cannot be expected that users of the tool provide manual code annotations about possible and impossible execution paths of the code. For example, when using a modeling tool like MATLAB/Simulink, the WCET analysis tool must be able to extract this information by analyzing the code generated by the code generator of MATLAB/Simulink.

2. The method must integrate into the development tool chain of customers without modification of tools from the tool chain (e.g., components of MATLAB/Simulink, code generator, C compiler).

   However, it may be possible to use the tool chain in a restrictive manner to enable the application of a certain WCET analysis method. For example, the available application development features of a modeling tool like MATLAB/Simulink may be restricted or certain compiler optimizations may be deactivated.

3. The method must be easily adaptable to new releases of software components of the tool chain. Expensive adaptations of the WCET method to new releases of software components have to be avoided.

   The situation that a development tool of the tool chain explicitly supports a specific WCET analysis methods is currently very rare. For example, it can be possible that a compiler provides certain support to perform WCET analysis [6]. But such tools are typically in a prototype state without commercial support. Therefore, the best current strategy for developing a WCET analysis tool is to adapt to existing COTS software development tools.

4. The WCET analysis method must be easy to retarget to different hardware settings, i.e., the implementation or configuration effort must be small enough for an economic useability of the WCET analysis method. Depending on the concrete WCET analysis method, there are in principle two different possibilities for retargetability. First, it can be required to order further implementation effort from the WCET tool provider. Second, it may be possible that the tool is flexible enough so that the customer can adapt the tool by himself. The latter approach is applicable for adequate measurement-based WCET analysis methods.

The adaption of a WCET analysis method to new hardware configurations can be kept easy when the WCET analysis method is based on measurements on the real hardware. Because in this case the WCET analysis method does not have to provide a so-called *exec-time model*, which describes the execution times for given code sequences. In measurement-based approaches the exec-time model is substituted by measurements on the real target hardware. There exist also measurement-based WCET analysis approaches that use hardware simulation instead of measurements on the real hardware [3, 4]. Such approaches rely on the existence of a cycle-accurate hardware simulator which is often not available.

In the following section a new WCET analysis method is presented that is able to fulfill the requirements from industry as given above. This WCET analysis method will be applied to program code automatically generated from MATLAB/Simulink models because there is additional information available about the structure of the generated code.

3 The WCET Analysis Framework

A new WCET analysis approach is needed to fulfill the requirements from industry listed in Section 2. Traditional methods based only on static code analysis are not flexible enough to retarget them with reasonable effort to new target processors. Though often used in practice, end-to-end runtime measurements are not an alternative, due to the exploding number of possible execution paths in real-size programs.

The WCET analysis method we describe in this paper is a hybrid approach of static and dynamic analysis methods. The dynamic part is performed by run-

time measurements on the real hardware platform. If available for the particular platform, the measurements could be also performed by a cycle-accurate simulator.

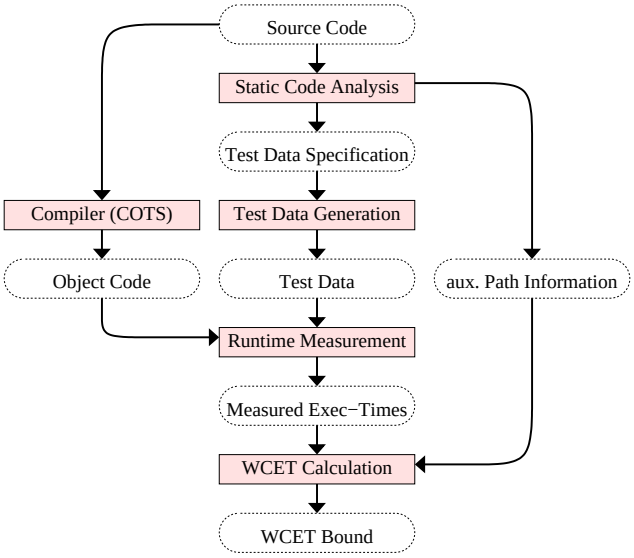

Figure 1. Components of the WCET Analysis Framework

The basic components of our WCET analysis framework are shown in Figure 1. The framework takes the program as input in both the source code and the object representation. The translation from source code to object code is done by a COTS compiler.

The *static code analysis* phase analyzes the source code with the goal to derive information about which test data should be generated for the runtime measurement and to derive path information that will be used by the final WCET calculation step:

Runtime Measurement is used to derive the instruction timing of paths through program segments. A *program segment* is a subgraph of a program's control flow graph with a unique start node such that only the start node has incoming edges from external nodes and all outgoing edges lead to the same external node.

The measurement of a specific execution path is enforced by generating input data that enforce the execution of this path. A coverage criterion has to be defined that describes the required runtime measurements. As already discussed, exhaustive execution path measurements of programs are not possible for real-size programs.

Having defined a coverage criterion, semantic code analysis is used to calculate the required test data. This analysis does not have to be implemented from scratch. Instead, the idea is to transform the program into a formal description of its program semantics that can be directly used by an existing analysis tool to generate the needed test data. The concrete analysis technique for test data generation has to be selected after evaluating its scalability regarding program size. Typical techniques that are interesting for this task are *model checking* [2, 9] or *constraint-based analysis* [10]. *Program slicing* [11] can be used to reduce the semantic models of the program by selecting only those parts of the code that influence the execution of a certain execution paths. Stepwise test data calculation can be used to further reduce the number of required test data. The idea is to calculate which further code locations will be also executed once input data for a specific code location have been selected. The measurements are done using a highly retargetable measurement framework.

To keep the test suite small, a hybrid approach consisting of static and dynamic WCET analysis is used. The *WCET Calculation* stage uses the execution time of each feasible path through program segments together with additional path information to calculate the WCET bound. The relevant path information includes iteration bounds for each loop, also called *loop bounds*. Depending on the code complexity, such loop bounds may be calculated automatically. If a loop bound cannot be calculated automatically, additional information has to be provided by the user. When analyzing code automatically generated from modeling tools like MATLAB/Simulink, additional knowledge about the structure of the code is known. As a result, most of the loop bounds in the generated code are typically hard coded and therefore can be derived automatically.

The *WCET Calculation* based on implicit path enumeration is done after performing the runtime measurements of the program segments [8].

The challenges of this WCET analysis framework are the automatic generation of test data and the extraction of control flow information from the program code. Both tasks cannot be done fully automatically for arbitrary program code. Therefore, user annotations respective restrictions on the code structure have to be used.

3.1 Decomposition of Execution Traces

To keep the number of required test data for runtime measurements within a feasible quantity, it is necessary to decompose the program paths into smaller parts and combine the obtained results to get the overall WCET bound. The choice of the right length of program subpaths for runtime measurement is based on a trade-off between complexity and precision. Complexity is given

by the number of required runtime measurements. In case of complex processors having an internal state that influences the execution time of instructions, measurement precision is better when measuring longer subpaths of the program.

One important aspect for the decomposition of execution traces is the demanded coverage criteria for the measurements. The coverage criteria will be defined at the level of *program segments*. For using this framework to obtain safe WCET bounds on a given hardware platform it has to analyzed what is the possible overestimation for particular coverage criteria.

4 Discussion

The design criteria of the measurement-based WCET analysis framework described in Section 3 are motivated by the requirements for an industrial-strength WCET analysis tool as summarized in Section 2. The decision of performing the program analysis at source code level is due to the requirement of high retargetability of the framework to new hardware platforms. However, for certain application domains it may be more important to have the analysis done after the code compilation at object code level. For example, it could be required to verify the path coverage calculated for source code level at object code level in case of critical code optimizations done by the compiler. In this case, the concept of the measurement-based WCET analysis using on automatic generation of test data is the same, but the implementation would be more hardware-dependent as it is also required to have a parser for the object code.

Technical realizations like inserting instrumentation code to measure the execution time of program segments are not discussed in this paper.

5 Summary and Conclusion

This paper describes a novel WCET analysis framework based on runtime measurements. The requirements for the framework are high portability to new target processors and an easy integration into COTS software development tool chains. We described a hybrid approach using static and dynamic timing analysis techniques. The central idea is to decompose the program paths into smaller subpaths and use formal methods to automatically derive the required test data to measure the execution time of the subpaths. Programs are structured into program segments to decompose program paths into smaller subpaths. After measuring the execution time of subpaths, a static WCET calculation is used to obtain the WCET bound.

Future work will focus on the assessment and selection of concrete formal program analysis techniques to generate the test data.

References

[1] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, Dec. 2002.

[2] A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the International Conference on Software Engineering*, Edinburgh, Scotland, UK, 2004.

[3] 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*, Hong Kong, Dec. 1999.

[4] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. International Conference on Computer-Aided Design (ICCAD '97)*, San Jose, USA, 1997.

[5] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[6] R. Kirner and P. Puschner. Timing analysis of optimised code. In *Proc. 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, pages 100–105, Guadalajara, Mexico, Jan. 2003.

[7] P. Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. 2nd Euromicro International Workshop on WCET Analysis*, Technical Report, York YO10 5DD, United Kingdom, June 2002. Department of Computer Science, University of York.

[8] P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *The Journal of Real-Time Systems*, 13:67–91, 1997.

[9] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. 8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)*, Washington DC, USA, Apr. 2001.

[10] N. T. Sy and Y. Deville. Consistency techniques for interprocedural test data generation. In *Proc. Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE03)*, Helsinki, Finland, 2003.

[11] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.