TRENDS IN TIMING ANALYSIS

Björn Lisper
Dept. of Computer Science and Electronics
Mälardalen University
P.O. Box 883
SE-721 23 Västerås
Sweden
bjorn.lisper@mdh.se

Abstract Static Worst-Case Execution Time (WCET) analysis aims to find safe upper bounds to the execution time of a program. We give a brief status report on the field of static WCET analysis, and we then present a personal perspective on the current and anticipated forthcoming trends in the area.

Keywords: Real-time System, Timing Analysis, Program Analysis

1. INTRODUCTION

A *Worst-Case Execution Time* (WCET) analysis finds an upper bound to the largest possible execution time of a computer program, or a time-critical part of a program. Reliable WCET estimates are crucial when designing and verifying embedded and real-time systems, especially safety-critical ones.

The WCET is often estimated through measurements. Estimates obtained in this way are, however, not reliable in general. An alternative is *static WCET analysis*, which determines a timing bound from mathematical models of the software and hardware. If the models are correct, then the analysis will derive a timing bound that is *safe*, i.e., greater than or equal to the true WCET.

In this paper, we discuss WCET analysis, its current status, and trends. What can be achieved today? How will trends in hardware and software affect the field? What is needed to turn WCET analysis into a widespread technique? This is a personal perspective, and we make no claims of completeness and scientific rigor.

The rest of this paper is organized as follows. In Section 2 we describe briefly what WCET analysis is, which basic approaches there are, and

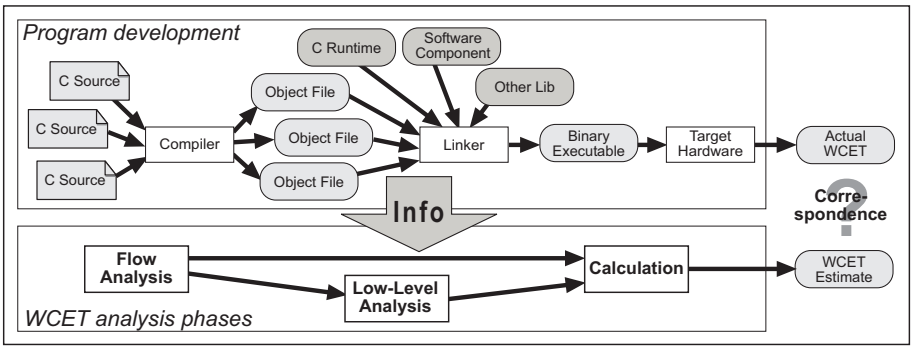

Figure 1. Embedded program development and static WCET analysis.

give a short account for the current status of the field. In Section 3 we discuss how trends in hardware architecture will affect WCET analysis. Section 4 provides a similar discussion as regards software, and requirements on WCET analysis. In Section 5, we monitor the research trends in WCET analysis. In Section 6, finally, we wrap up.

2. WCET ANALYSIS

Static WCET analysis is usually divided into three phases: a (fairly) machine-independent *flow analysis* of the code, where information about the possible program execution paths is derived, a *low-level analysis* where the execution times for sequences of instructions are decided from a performance model for the target architecture, and a final *calculation* phase where the flow and timing information are combined to yield a WCET estimate. See Figure 1.

The purpose of the flow analysis phase is to find constraints on the possible execution paths of the program, like upper limits to loop iterations, infeasible path constraints, etc. *Automatic* flow analysis calculates this information with little or no manual intervention. It is in general impossible to derive exact information. Automatic methods therefore calculate approximate flow information, and allow additional information as *manual annotations* (3, 13, 14).

Flow analysis can be done on source code, on compiler-intermediate code, or on binary code. In the first two cases, the flow information must somehow be mapped to the binary code. Approaches include abstract interpretation to bound the values of execution counters (7), pattern-matching (8, 24), symbolic execution (17), and the use of Presburger Arithmetics to identify loop parameters (3).

Today, most processors use performance-enhancing features such as *pipelines*, *caches*, *branch predictors*, and *instruction-level parallelism*.

These features unfortunately yield complex timing models. Low-level analysis research mostly concerns methods to deal with such timing models.

The timing effects of pipelines are mostly local, and can be handled quite well (4). Caches are harder to analyze, since they yield a global, history-dependent timing model, where memory access times depend on the current contents in the cache. A *cache analysis* attempts to predict the contents of the cache (6). Instruction streams often are quite easy to predict, and thus instruction caches can often be handled quite well, whereas data caches often are subject to more dynamic memory accesses, and thus are harder to analyze. Branch predictors, like caches, yield history-dependent performance models, and low-level analyses for branch predictors attempt to estimate their states (2).

Modern high-end processors use instruction-level parallelism, often through a superscalar architecture. Such architectures are hard to analyze w.r.t. timing properties, due to their dynamic instruction scheduling. Nevertheless, attempts have been made (15).

The final step in WCET analysis is to calculate a WCET estimate from flow and timing information. Three common methods are *tree-based calculation* (18), *path-based calculation* (8, 23), and *IPET* (Implicit Path Enumeration Technique) (14, 21). IPET is the most common calculation method today, and it calculates a WCET bound by solving an integer linear programming problem.

WCET analysis is currently taking the step from research to industrial use. Three commercial tools exist, which are mainly used in automotive and avionics industry to analyze hard real-time systems. The technique is, however, not yet widely adopted.

For reasonably simple embedded processors, with pipeline but without cache, the current tools can provide WCET estimates which are typically 5–10% larger than the largest measured execution time. However, most programs today require extensive manual annotations, constraining the program flow, to get there. The annotations often require deep understanding of the code, and they are cumbersome to make. More complex processor architectures can also be handled, but for a more limited set of programs, and analysis times grow rapidly. The practical limit today seems to be at the level of processors like Motorola ColdFire 5307 and PowerPC 755 (24).

3. HARDWARE

We see a trend towards even more complex processor architectures, with more advanced features enhancing the overall performance. Em-

bedded systems tend to migrate towards more complex processors. Therefore, future WCET analysis methods must be able to keep up with the development of high-performance architectures.

There is an inherent conflict between analyzability and modern performance-enhancing features. These features typically make the processor adapt dynamically to run the code at hand faster, through caches, branch predictors, etc. They typically record the execution history into a complex internal processor state. The more complex the internal state, the harder the timing analysis. Not only does it become harder to estimate the WCET with good precision, the difference between the true WCET and the average execution time also tends to increase.

On the other hand, a trend in embedded systems is to put more of the architecture under software control, in order to cut corners in chip cost, and to allow for smart optimizations. Two examples are *software-controlled cache locking*, and *scratchpad memories*. Both allow a more explicit handling of on-chip memory, which is beneficial for analyzability (26, 25).

Unwanted interaction between hardware features can hurt the analysis. A modularized WCET analysis is often less costly. For instance, the analysis is simplified if the cache behaviour can be analyzed separately, to provide information about hits and misses, which then is used in the further low-level analysis. If more misses always implies longer execution time, then the cache analysis can safely assume that a memory access is a miss if it is not surely found to be a hit, which makes the analysis less complex. However, for superscalar processors, a cache miss may yield a shorter execution time (17). This is due to the dynamic instruction scheduling, where a delayed release of an instruction in the end may give a better schedule. Unwanted interactions of this kind necessitate a more integrated low-level analysis, which can be very detrimental to performance since the sets of possible hardware states grow rapidly.

Another kind of unwanted interaction is due to *sharing of resources which have a stateful timing model*. For instance, if a processor has separate instruction and data caches, then the analysis of one cache may be precise even if the other is not. However, some processors have a shared cache: a poor cache hit/miss estimation for one of the access streams will then typically pollute the information about the whole cache contents, even if the other access stream could have been accurately analyzed. As a result, the total cache analysis will be less precise.

A very important paradigm shift in processor architecture is the introduction of *multicore processors*, where a single chip hosts a multiprocessor. High-end PC's already have dual-core Pentiums, and in a few

years multicore processors are likely to prevail. This changes the rules of the game radically.

On the positive side, the processor cores will be simpler than current high-performance processors. Thus, that part of the low-level analysis will be simpler. On the other hand, the processors will often have shared resources, like buses, and cache memories. Since different threads will access these resources, the concurrency will lead to a combinatorial explosion of possible state transitions, which then yields both long analysis time and poor precision.

A big problem is that all current WCET analysis methods assume a strictly sequential execution model. In general, the real-time research community seems to assume that parallel hardware is to be utilized by sequential tasks running on different processors. This assumption is very doubtful: computationally heavy tasks are advantageous to parallelize, and most likely this will happen. WCET analysis for parallel programs must then be developed in order to analyze such tasks.

4. SOFTWARE AND REQUIREMENTS

Traditionally, time-critical embedded applications have been programmed in C or assembler, and current WCET analysis methods mostly target C or code generated from C. This is imperative code, which often has a reasonably simple structure. Such code is relatively easy to analyze. Whole programs are typically analyzed, which means that the tool has full information about the code.

However, there is a migration to higher levels of abstractions. *Higher level programming languages*, such as object-oriented (OO) languages, become more used. *Model-based design*, where code is generated from models, is rapidly gaining ground in many application areas. *Component-based software engineering*, where program components are reused and combined, is also a strong trend.

Higher-level programming languages give rise to new problems. The control structure typically becomes more dynamic. For OO languages, with methods rather than functions, methods are accessed indirectly through a method table. This makes it harder to reconstruct the control flow. Data structures also tend to be more dynamic, and memory management might be automatic. It is then harder to decide the adresses of memory accesses, which makes it much harder to predict the memory access times.

Some high-level languages, like Java, are implemented on virtual machines. These machines introduce an additional abstraction layer between the program and the hardware, which makes analysis harder. If

the implementation uses just-in-time compilation, then it will be very hard to predict the actual instructions executed. WCET analysis of JVM code has been attempted (19), but the results have not been too encouraging.

However, many programs written in high-level languages do have a quite static structure. The abstraction mechanisms may be used to support reuse over different static configurations, rather than for truly dynamic run-time behaviour. For such programs, program analyses may be able to uncover the static structure, as well as other properties like sizes of computed data structures (9). Program specialization techniques, such as *partial evaluation* (10), may also yield more analyzable (and also faster) programs. Declarative languages are particularly amenable to program analysis and transformation, due to their clean semantics.

Model based design, where code is generated from models, poses new challenges and possibilities as regards WCET analysis. Since different tools, for different purposes, generate very different code, the challenges will be very tool-specific. For instance, code generated for a state machine is likely to be highly unstructured, whereas control system code generated from a simulation model should have a more regular structure. The way a tool chooses to implement a model feature, like a FSM, can also have a great impact on the WCET analyzability.

Obviously, WCET analysis of generated code should use tool-specific information about how the code is generated whenever possible. It may, for instance, be the case that a tool generates code where the loop iteration bounds are directly related to the size of a matrix in the model. Annotations bounding the loop iterations can then be automatically generated from the model. Experiments in this direction have been done for Simulink (12).

Component-based software engineering (CBSE) is gaining ground. It emphasizes the structuring of software into reusable components, with well-defined interfaces. There are many different component models, ranging from quite static models, where components are statically configured, to very dynamic models where components can be swapped at runtime.

A key feature is communication. Since the component model typically is independent of the component implementation language, and even the host processor, the communication often includes marshaling of data between different formats. This is especially noticeable for distributed component models, where middleware like CORBA is common.

There is an increasing interest in CBSE for embedded systems. This raises the issue of how to analyse such software w.r.t. timing. Again, there are a number of problems. First, components may be "black

boxes", with little information about the code. Second, reusable components may be heavily parameterized. If the WCET is parameter-dependent, then a single upper bound may be very untight. Third, complex communication protocols may make the component communication very hard to analyze.

These problems must be solved to make WCET analysis useful for component-based software systems. Conversely, component models for hard real-time systems should be designed to be analyzable.

Little attention has been paid to the role of WCET analysis in the *software development process*. The current tools require compiled binaries, and can thus be applied only late in the development process. However, timing-aware software design is often done with *time budgets*, where different software parts are given maximal execution times. It is then very interesting to estimate the execution time early, before the complete binaries are available, to help set up these budgets. Thus, there seems to be a market for early, approximate WCET analysis, using, say, incomplete source code and crude performance models.

For high performance processors, the current WCET estimates tend to be very pessimistic compared with the average execution time. Many applications, especially in areas like telecommunications and multimedia, have Quality of Service (QoS) requirements. Infrequent deadline violations are then not harmful, and some violations are typically allowed in order to utilize the hardware better. For such systems, it would be more appropriate to derive statistical estimates for violations of given deadlines, rather than absolute upper WCET bounds. A step in that direction is the pWCET framework for probabilistic WCET analysis (1).

5. WCET ANALYSIS TECHNIQUES

What developments can be expected in WCET analysis? We believe that there is a need for approximate WCET analyses, with some kind of probabilistic guarantees. This is due to the demands from the large set of QoS-oriented applications, as well as the wish to introduce timing analysis early in the tool chain. We also believe that hybrid analyses, involving both static analysis and measurements, will be further developed. Such analyses can avoid the costly and pessimistic low-level analysis, at the price of not obtaining absolutely safe upper WCET bounds. For single-path programs, measurements can give safe and accurate WCET estimates, and a single-path programming style has been advocated (20). The aforementioned probabilistic WCET analysis (1) also includes measurements.

Traditional, static low-level analysis is also developing. A very interesting trend is the incorporation of methods from model checking to handle very large state spaces. For instance, BDD's have been proposed (27).

WCET-aware compilation attempts to improve the WCET, or its analyzability, rather than average performance. An example is dynamic cache locking to keep the cache contents predictable (25).

An important aspect of WCET analysis tools is their *usability*. Current tools require many manual annotations, in particular to constrain the program flow. To give flow information manually is cumbersome, and requires a deep understanding of the code. This restricts the usability of the tools (5). The level of automation must be raised, which requires better methods in automatic flow analysis.

However, to find tight flow information is difficult. Loop bounds may depend in complex ways on inputs, pointers, etc. Flow analysis of binary code is especially difficult. If the compiler can map flow information from the source code to the binaries, then this problem is alleviated. In particular, manual annotations are best given on source code level, and they cannot be completely avoided in general. Studies in this direction have been made (11), but production compilers must adopt this kind of technique if it is to have any impact on real practice.

Another option is *parametric* WCET analysis (16). Such an analysis computes a formula for the WCET bound rather than a single number. Code with input-dependent WCET is common in many applications (22). A parametric analysis can also help analyze the sensitivity of the WCET w.r.t. different parameters, which is useful when developing code with time budgets.

6. CONCLUSIONS

WCET analysis is a maturing technology that is currently being introduced in industry. However, it is not widely established yet. While promising and important, many challenges remain to meet before the technique will be adopted on a wider basis. The most important challenges are to keep up with the hardware development, to increase the level of automation of the analysis, and to widen the scope to include also soft real-time systems with QoS requirements.

The ability to meet these challenges depends on hardware, compilers, and other tools which generate code. If one were to make a wish list, it would include: WCET-aware compilers, with the ability to map program flow information from source code to binary code, and hardware which avoids shared resources and hidden stateful features to improve average

performance. For future multicore processors, we wish for mechanisms to partition shared resources, like caches or scratchpad memories, between tasks.

ACKNOWLEDGMENTS

I want to thank Jan Gustafsson, Andreas Ermedahl, and Christer Sandberg for their valuable comments.

REFERENCES

[1] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd IEEE Real-Time Systems Symposium*, 2002.

[2] François Bodin and Isabelle Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proc. 17th Euromicro Conference of Real-Time Systems*, pages 33–40, July 2005.

[3] Bound-T tool homepage, 2006. `www.tidorum.fi/bound-t/`.

[4] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, April 2002.

[5] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Experiences from industrial WCET analysis case studies. In Reinhard Wilhelm, editor, *Proc. 5th Int. Workshop on Worst-Case Execution Time Analysis*, pages 19–22, Palma de Mallorca, July 2005.

[6] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17:131–181, 1999.

[7] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *Proc. 10th IEEE Int. Workshop on Object-oriented Real-time Dependable Systems*, Sedona, USA, February 2005.

[8] C. Healy, R. Arnold, Frank Müller, David Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[9] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[10] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Hertfordshire, UK, 1993.

[11] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Austria, 2003.

[12] Raimund Kirner, R. Lang, G. Freiberger, and Peter Puschner. Fully automatic worst-case execution time analysis for matlab/simulink models. In *Proc. 14th Euromicro Conf. of Real-Time Systems*, pages 31–40, June 2002.

[13] Raimund Kirner and Peter Puschner. Transformation of path information for WCET analysis during compilation. In *Proc. 13th Euromicro Conf. of Real-Time Systems*, pages 29–36, Delft, June 2001. IEEE Computer Society Press.

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

[15] S. Lim, J. 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*, December 1998.

[16] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In Jan Gustafsson, editor, *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis)*, pages 77–80, Porto, July 2003.

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

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

[19] Peter Puschner and Guillem Bernat. WCET analysis of reusable portable code. In *Proc. 13th Euromicro Conf. of Real-Time Systems*, pages 45–52, Delft, June 2001.

[20] Peter P. Puschner and Alan Burns. Writing temporally predictable code. In *Proc. 7th IEEE Int. Workshop on Object-oriented Real-time Dependable Systems*, pages 85–94, San Diego, January 2002. IEEE Computer Society.

[21] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, July 1997.

[22] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static Timing Analysis of Real-Time Operating System Code. In *Proc. 1st Int. Symposium on Leveraging Applications of Formal Methods*, October 2004.

[23] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proc. 4th Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, November 2001.

[24] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD thesis, Saarland University, 2004.

[25] Xavier Vera, Björn Lisper, and Jingling Xue. Data Cache Locking for Higher Program Predictability. In *Proc. Int. Conf. on Measurement and Modeling of Computer Systems*, pages 272–282, San Diego, CA, June 2003. ACM Press.

[26] Lars Wehmeyer and Peter Marwedel. Influence of onchip scratchpad memories on WCET prediction. In Isabelle Puaut, editor, *Proc. 4th Int. Workshop on Worst-Case Execution Time Analysis*, pages 15–18, Catania, June 2004.

[27] Stephan Wilhelm. Efficient analysis of pipeline models for WCET computation. In Reinhard Wilhelm, editor, *Proc. 5th Int. Workshop on Worst-Case Execution Time Analysis*, pages 19–22, Palma de Mallorca, July 2005.