Approximate Worst-Case Execution Time Analysis for Early Stage Embedded Systems Development

Jan Gustafsson[†], Peter Altenbernd[⋆], Andreas Ermedahl[†], Björn Lisper[†]

[†]School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden. {jan.gustafsson,andreas.ermedahl,bjorn.lisper}@mdh.se
[⋆]Department of Computer Science, University of Applied Sciences, Darmstadt, Germany. p.altenbernd@fbi.h-da.de

Abstract. A Worst-Case Execution Time (WCET) analysis finds upper bounds for the execution time of programs. Reliable WCET estimates are essential in the development of safety-critical embedded systems, where failures to meet timing deadlines can have catastrophic consequences.

Traditionally, WCET analysis is applied only in the late stages of embedded system software development. This is problematic, since WCET estimates are often needed already in early stages of system development, for example as inputs to various kinds of high-level embedded system engineering tools such as modelling and component frameworks, scheduling analyses, timed automata, etc. Early WCET estimates are also useful for selecting a suitable processor configuration (CPU, memory, peripherals, etc.) for the embedded system.

If early WCET estimates are missing, many of these early design decisions have to be made using experience and "gut feeling". If the final executable violates the timing bounds assumed in earlier system development stages, it may result in costly system re-design.

This paper presents a novel method to derive approximate WCET estimates at early stages of the software development process. The method is currently being implemented and evaluated. The method should be applicable to a large variety of software engineering tools and hardware platforms used in embedded system development, leading to shorter development times and more reliable embedded software.

1 Introduction

Embedded systems are special-purpose computer systems designed to perform one or a few dedicated functions, often with timing constraints. They are usually embedded into devices including hardware and mechanical parts.
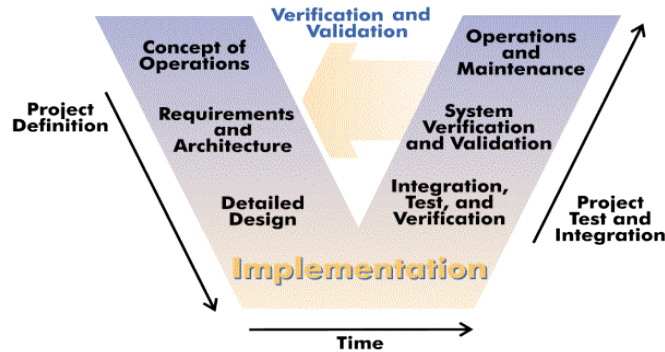
Fig. 1. The V-Model (graph from [1]).

Different types of (and even parts within) embedded systems may have different timing requirements. Some safety-critical embedded systems have hard real-time requirements, i.e., failures to meet timing deadlines can have catastrophic consequences. For these, a safe (i.e., never underestimated) WCET of the software is a key measure.

1.1 The need for early WCET estimates

The specific properties of real-time embedded systems puts certain demands on the development of such systems. Often, hardware and software are developed in parallel (hardware-software co-design). Since embedded systems often are in a high-volume market, it is important for those systems to choose a suitable processor configuration (CPU, memory, peripherals, etc.) which is just powerful enough, in order not to use a too costly hardware. Thus, embedded systems have smaller resource margins than, e.g., desktop computers, and special attention has to be given to the timing of the software. Consequently, it is important to assess the worst-case timing (i.e., the WCET) of the software to be able to choose a suitable processor configuration.

There are several software development models to be used for embedded real-time systems, e.g., the traditional waterfall model (Requirements – Design – Development – Testing), or the more advanced V-Model [1] shown in conceptual form in Fig. 1. In the V-model, the requirements, architecture and design in the left part of the model will be verified in the activities in the right part of the model (integration, test, verification, system verification and validation). This is also true for the timing domain of the system. For example, the timing requirements defined early in the model (on the left side) will be verified against the system on the same level in the model on the right side. Software timing decided during module design will be tested during module test etc.

Early WCET estimates would be useful in the early stages of real-time embedded systems development (like the requirements, architecture and design stages in the V-model) for a number of reasons. Modern embedded system development typically includes a large variety of software engineering tools, such as modelling and component frameworks, schedulability analysis, timed automata, etc. The tools are used to, e.g., decide how tasks should be generated from larger

software components or models, how to distribute tasks to computer nodes, what hardware to use on the different nodes, what priorities to assign to different tasks, etc. To be able to model, validate and verify the real-time properties of early sketches of the system, these tools need to associate some type of execution time bounds to their inherent high-level code constructs.

Existing WCET analysis methods often fail to provide early stage WCET estimates. The main reason is that the following requirements must be fulfilled, which is often not the case at the early stages of the project:

– the program to be analysed must be compiled and linked to an executable binary, and
  – either a useful input data set must be available, and the actual hardware (or a timing accurate simulator) must be available in a setup that allows for correct measurements, or
  – a (safe) timing model of the actual hardware must be available.

See further Section 2.

For modern embedded system development this is a problem, since as described above, good approximate WCET estimates are needed already during early system development. Moreover, if the WCET bounds derived on the executable program violates the timing bounds assumed in earlier development stages, it may result in costly system re-design, including code re-implementation, re-testing and re-analysis. It is a well-known fact that the cost for correcting functional errors and doing larger system changes becomes higher at later stages of the development process [2,3]. There are all reasons to believe that this cost curve also holds for errors in the timing domain.

1.2   Paper contribution and content

In this paper, we will present an analysis method to provide early approximate WCET estimates. The method consists of two main steps; 1) *timing model identification using a set of programs*, and 2) *WCET estimate calculation using flow analysis and the derived timing model*.

Our method is not guaranteed to always provide safe WCET bounds. However, we believe that during early phases of system development it is often sufficient to provide *reasonably accurate WCET estimates*. Such estimates will allow the system designer to make a good initial system model, where the approximate WCET estimates of different code parts are taken into account. At later development stages a more precise WCET analysis tool can be used, allowing the rough WCET estimates and the initial system model to be refined. The main advantages of our method, when compared to existing WCET analysis methods, are the following:

– no timing model of the actual hardware has to be available; it will be created in the model identification step.
– the actual hardware/simulator only has to be available during an initial timing model identification step.
– model identification only has to be performed once for each compiler/hardware combination.

– the program for which a WCET estimate will be calculated does not have to
be executable; it does not even have to be compiled.

The rest of this paper is organized as follows. Section 2 gives an overview
of WCET analysis methods. Section 3 gives short descriptions of related work
in the area. Section 4 presents the proposed method. Section 5 describes the
implementation that we are currently working on. Finally, in Section 6, we discuss
the approach and give ideas for evaluation and further research.

2   Worst-Case Execution Time Analysis

A *Worst-Case Execution Time* (WCET) analysis finds an upper bound to the
worst possible execution time of a program. WCET analysis must handle the fact
that a program typically has no fixed execution time. Different execution time
may occur for different inputs, due to the characteristics of the software, as well
as of the computer upon which the software is run. Thus, both the possible inputs
as well as the properties of the software and the hardware must be considered
in order to predict the WCET of a program. An overview of WCET analysis
is given in [4]. Here, we will give just a short introduction of the three main
approaches used: *measurements*, *static timing analysis*, and *hybrid analysis*.

Measurements, also known as *dynamic timing analysis*, is the traditional way
to determine the timing of a program. Basically, the program is executed many
times with different inputs and the execution time is measured for each test run.

For WCET analysis, there are problems connected with measurements: The
methods often means labor-intensive and error-prone work, and even worse, it
cannot guarantee that the WCET has been found. This is because each mea-
surement exercises only *one* path. For most programs, the number of possible
execution paths is immense, and therefore too large for exhaustive testing. Also
it is, in general, very hard to find the worst case input. This means that the
set of inputs may not include the worst case path, and thus the method may
underestimate the WCET. WCET measurements require:

– the program to be analysed can be compiled and linked to an executable
binary. This requires that the program is "finished" in some sense (it must
compile and link without errors, and execute to completion),
– an input data set which covers all of the program paths (or as many as
possible, hopefully including the WCET path) is available, and
– the actual hardware (or a timing accurate simulator) is available in a setup
that allows for correct measurements.

Static timing analysis estimates the WCET of a program without actually run-
ning it. The analysis avoids the need to run the program by simultaneously
considering the effects of all possible inputs, including possible system states,
together with the program's interaction with the hardware. The analysis relies
on mathematical models of the software and hardware involved. Given that the
models and the calculation never make underestimations, the result is a *safe*
timing estimate that is greater than or equal to the actual WCET.

Static WCET analysis is usually divided into three phases: a *flow analysis* where information about the possible program execution paths is derived, a *low-level analysis* where the execution time for atomic parts of the code (e.g., instructions, basic blocks or larger code sections) is decided from a model of the target architecture, and a final *calculation* phase where the derived flow and timing information are combined into a resulting WCET estimate.

Due to the complexity of today's software and hardware, both flow- and low-level analysis may result in over-approximations, e.g., reporting too many paths as feasible or assigning too large timings to instructions. Thus, the calculation will give a safe but potentially pessimistic WCET value. Static WCET analysis requires:

– that the analysed program can be compiled and linked to an executable binary, and
– that a (safe) timing model of the hardware is available, something that can be very hard and costly to provide.

Depending on the type of static analysis, more requirements may be added. If the static analysis is based on binary code, a binary decoder must be developed for the used binary format in order to re-generate the control flow graph of the program. This flow graph is used by the static analysis.

Other tools do flow analysis on source code level or intermediate code level. For these tools, some kind of compiler support is necessary.

A possibility to add manual annotations for, e.g., flow constraints, is useful as a complement to the flow analysis. This may reduce the overestimation of the analysis. Some tools use annotations to describe input data limits for the same purpose.

Hybrid analysis techniques combine measurements and static analyses techniques. The tools use measurements to extract timing for smaller program parts, and static analysis to deduce the final program WCET estimate from the program part timings. Since measurements are used, the resulting WCET estimates are not guaranteed to be safe. Hybrid WCET analysis requires:

– that the analysed program can be compiled and linked to an executable binary,
– an input data set which covers all of the program paths (or as many as possible, hopefully including the WCET path) is available, and
– the actual hardware (or a timing accurate simulator) is available in a setup that allows for correct measurements.

3 Related Work

The idea of early timing analysis has been explored in a number of variants. The TimingExplorer [5] developed by AbsInt [6] is an assisting tool to find early timing estimates to be able to decide suitable hardware architectures for real-time systems. The available source code is compiled and linked for each of the cores in question. The user also has the possibility to specify different memory layout and cache properties. Each resulting executable is then analysed with the TimingExplorer to get a WCET estimate for the chosen core and hardware

architecture. TimingExplorer uses a run-time optimised analysis to quickly produce WCET estimates. It should be noted that TimingExplorer trades precision against, e.g., speed. It is also not certain that the estimates are safe.

Even though the goals and the achievements of the TimingExplorer approach are similar to ours, there are some differences: TimingExplorer requires a hardware model, something which is not needed in our approach. Also, the source code for the programs for which WCET estimates are to be found does not have to be compiled and linked in our approach. This means that it probably can be applied easier and earlier than TimingExplorer. On the other hand, TimingExplorer will probably give more precise WCET estimates than our method.

Another interesting approach for early stage WCET analysis is the integration of the aiT WCET analysis tool [6] into the SCADE [7, 8], and ASCET [9] software tools. The use of these tools gives the possibility to generate code, and possibly WCET estimates, early from sketches of the system.

Their approach requires support from different software tools for keeping a mapping between the high-level code constructs and the corresponding binary code snippets. Moreover, aiT's low-level analysis must have been ported to the used target processor. Thus, the approach should be less portable than ours. Moreover, it can only be applied rather late in the system development process, since aiT requires the full binary program to be available for its WCET analysis. No flow analysis is performed on the high-level code level.

Lisper and Santos has recently developed a new kind of regression method, based on end-to-end measurements of programs, where the resulting timing model is guaranteed to not underestimate any observed execution times [10]. This is similar to the model identification used in our method. In contrast to our method, the Lisper/Santos method works completely on the binary level.

There are some earlier efforts to do early stage WCET analysis upon Petri nets [11], Statechart models [12], and Matlab/Simulink models [13]. Compared to our approach, all these approaches are less portable and require much support from the high-level development tool.

There have been approaches to WCET analysis on Java. Persson et al. [14] assign timing costs to Java constructs using an attribute grammar. Bernat et al. [15] do analysis on the Java Byte Code (JBC) level and provide a mechanism for giving compiler/language independent WCET annotations in the Java source code. Bate et al. [16] extended the latter approach to also include a timing model for JBC instructions. Compared to our approach, all these approaches rely on manual flow annotations and manual assignments of times to constructs.

Bartlett et al. [17] use program traces to derive parametric upper loop bounds for WCET analysis. Their work, and the measurement-based approach to WCET analysis used by Rapita systems [18], both have similarities to our idea of deriving a timing model by program runs.

Franke [19] uses measurements and regression analysis to derive a reasonably timing-accurate simulator for assembly code instruction sets, thus having some similarities to our timing model derivation approach.

4 Early Stage WCET Analysis

Our method is based on the same main phases as ordinary static WCET analyses; i.e., a *flow analysis*, a *low-level analysis*, and a final *calculation* phase. In particular, the method combines a worst-case oriented flow analysis with the usage of an approximate timing model for low-level analysis, both working on code constructs available during early phases of embedded system development.

The method is based on the assumption that the possible flows through a program is similar independent of which code level the program is represented in. For example, an upper limit on the number of times a certain code construct is taken in the high-level code is normally mimicked as a limit on the corresponding low-level instructions. However, a direct mapping between the high level code and the low-level code is not always possible, due to compiler optimisations.

The timing behaviour of a program on different code levels is more complex. It is often not possible to assign constant timing to high-level code constructs due to the timing variability caused by different hardware features. Moreover, compiler optimisations may make the mapping between different code levels very hard. Instead, our method derives an *approximative timing model* for high-level code constructs by systematic measurements.

As mentioned in Section 1, our method for obtaining early stage WCET estimates includes two main steps: 1) *Timing model identification using a set of programs*, and 2) *WCET estimate calculation using flow analysis and the derived timing model*.

4.1 Timing model identification

We propose a timing model identification using a set of programs to derive an approximate (but not guaranteed safe) timing model for different code constructs found in program code. The code level to be used for the model identification, hereafter called *timing model code level (tmcl)*, can be on any level of choice. It can be the high-level code itself, or some code possible to generate from the high-level code in one or more steps. For example, it can be a modelling language, C, C++ or some other source code format, intermediate code (maybe emitted by a compiler or some modelling tool), or even assembler.

The idea is that by identifying a timing model for the tmcl code, the WCET analysis can be performed directly on this code, rather than analyzing the compiled binary and a precise timing model on the binary level. This eliminates the need to compile the code for the target system. The higher the level of the code, the earlier the analysis can be done, but on the other hand the WCET estimates are likely to be less precise.

The timing model identification is made by a combination of *measurements* and *regression analysis* as described in the following. We propose to identify a linear timing model
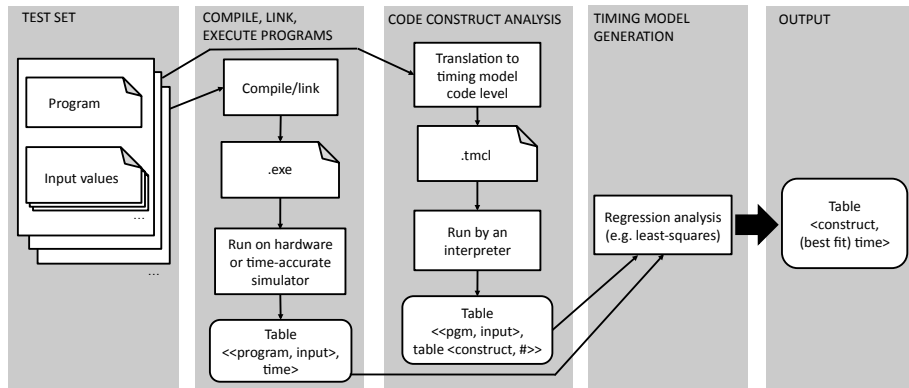
$$t = \sum_{i=1}^{n} c_i \tau_i$$

Fig. 2. Timing model identification.

for *n code constructs*. These can, for instance, be arithmetic/logic operators, program variable accesses, statements altering the program flow, or possibly more complex constructs that recur often enough. The model assumes a constant execution time τ_i to each construct i, c_i is the number of times the construct is executed, and t is the execution time predicted by the model. The accuracy of such a simple model will depend on how closely the selected code constructs correspond to (fixed sequences of) instructions in the corresponding binary, compiled for the target machine. The model will typically be derived for a given combination of target hardware, compiler, and possibly also compiler options (like optimisation level). The model identification selects the execution times τ_i to fit the model as well as possible to a set of given runs on the target machine.

An overview of timing model identification is shown in Fig. 2. First, a test set of programs and corresponding sets of inputs is selected and executable binaries are generated using the compiler of interest. For each pair of input and program, two runs are made (similar to [10]):

— the binary is run, and the actual execution time is recorded, and
— the tmcl code is run, with the same inputs, by an interpreter or similar that records the number of times each code construct is executed.

For each such (double) run j we obtain an actual execution time t_j, and, for each code construct i, a number of executions c_{ij}. According to the linear timing model, we obtain an equation

$$t_j = \sum_{i=1}^{n} c_{ij}\tau_i$$

If we make m such runs, we obtain a linear equation system $\boldsymbol{t} = C\boldsymbol{\tau}$ where the coefficients in the $m \times n$-matrix C are the recorded execution counts c_{ij}.

If the timing model happens to be exact, then the system has a unique solution. Most likely it is not. Then, if the number of linearly independent row vectors in C is at least n, we can perform some kind of *regression* to find a "best" solution $\boldsymbol{\tau}^*$ that in some sense minimizes the deviation of the execution times predicted by the model and the actual execution times. A possible choice is the
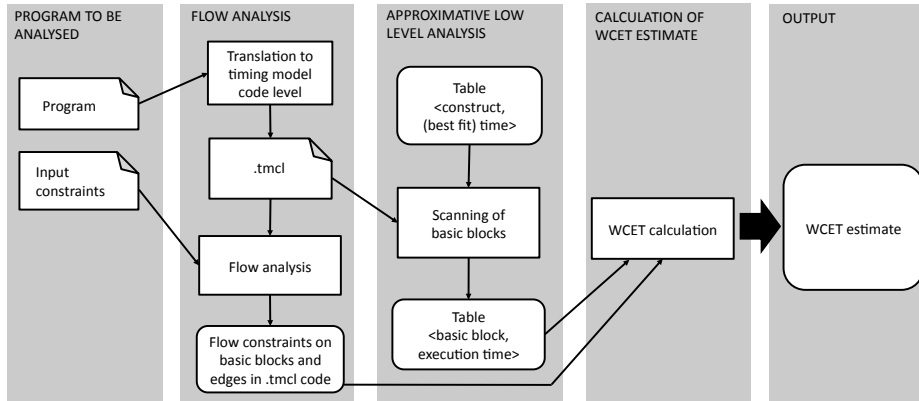
Fig. 3. Early stage WCET estimate calculation

usual linear regression, or least-squares method [20]. Another choice is the *max regression* proposed in [10], which has the property that the model never will underestimate any observed execution times.

The resulting timing model will depend both on how the source code is compiled to executable code, e.g., on the compiler and its options, on the translation to tmcl format, as well as on the characteristics of the different hardware features, such as caches and pipelines of the used hardware. Ideally, if we have a 1-1 mapping between the high level code and the binary code, and if the code has basic blocks with constant execution times, we may get a very good fit. When this is not the case, the fit will be less good, but probably still good enough to give a WCET estimate that is helpful in early design stages.

4.2 Early stage WCET estimate calculation

The approximate WCET for the program of interest will be derived using flow analysis, the approximate timing model derived in the model identification step, and a final WCET calculation. Neither the executable of the program, nor the hardware/simulator is required during this step (see Fig. 3). Note that the derived WCET for the program is valid for the combination of target hardware, compiler, and compiler options that was used in the model identification step.

First, the analysed program is translated to tmcl format. We then perform a flow analysis of the program, using the tmcl representation of the program and (optionally) constraints on input variables to the program. The flow analysis generates flow constraints on basic blocks and edges in the tmcl code.

The approximative low-level analysis uses the tmcl form of the program and the table of the best possible fit of timing costs to the different code constructs (generated in the model identification step). While scanning through the program, execution time estimates for basic blocks will be generated and saved.

Finally, we can derive a WCET estimate using some existing WCET calculation method, see e.g., [4], combining the results of the previous steps.

5 Implementation

The method described in this paper is currently being implemented by the WCET group at Mälardalen University [21] in the SWEET (SWEdish Execution time Tool) WCET analysis tool. The ALF (ARTIST2 Language for WCET Flow Analysis) language [22] has been selected as the tmcl format.

ALF is a language intended for flow analysis for WCET calculation. SWEET includes an ALF interpreter, which output statistics of ALF construct exercised during a program run. Thus, the timing model identification will assign timing to different constructs found in the ALF code. To perform the measurement runs, we plan to test both some clock-cycle accurate simulators and some hardware equipped with time-measurement facilities.

We will also do the flow analysis on ALF, using SWEET's powerful flow analysis methods [23–25]. The result will be given as constraints on the number of times different basic blocks of the ALF code can be executed and edges of the control flow graph can be followed.

The final calculation of the WCET estimate will be made using one of SWEET's different calculation methods [26].

6 Discussion and Future Work

The method will be evaluated using WCET benchmarks and industrial code. This first version of the method uses linear equations which are exact only if the tmcl code constructs always have the same corresponding code in the binary representation, and that the high code constructs always have the same execution time. We will in, a systematic way, study the consequences of deviations from these requirements. The experiences we will get from the evaluation will hopefully give some ideas on how to extend the method. There are a number of interesting ideas to explore, and some of them has been mentioned in the paper. Other ideas include:

— *More precise model generation.* It may be possible to do a context sensitive and more precise model where ALF constructs have different costs depending on execution context, e.g., whether it is the first loop iteration or not that is executed, since cache effects can give different execution times for the corresponding "real" instructions in the binary.

— *How will compiler optimisations and other code transformations affect the accuracy?* Code transformations make the mapping between the high-level code constructs and the binary program more complex.

— *How may different hardware features affect the accuracy?* Modern processors use many performance-enhancing features, like caches, pipelines, branch-prediction, out-of-order execution, etc., all which may cause the execution time for instructions to vary. How well will timing effects of such types of features be captured by the resulting timing model? Are some types of hardware features more troublesome than others?

— *Selection of model identification sets.* How many programs and input data sets are needed for getting an accurate timing model? How similar must the

test programs used to generate the timing model be to the program for which WCET estimates will be derived?

– *How to handle a mix of source code formats?* An embedded system project may involve a variety of source code formats, including different high-level languages, C or C++ code, or even assembler. An interesting property of our implementation is that if programs can be translated to ALF, they are analysable by the method. We expect a number of formats to be able to translate to ALF.

References

1. Wikipedia: V-Model (2009) `http://en.wikipedia.org/wiki/V-Model`.
2. Boehm, B.W.: Software Engineering Economics. Prentice Hall PTR, Upper Saddle River, NJ (1981)
3. Westland, C.J.: The cost of errors in software development: evidence from industry. Journal of Systems and Software **62** (2002) 1–9
4. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution time problem — overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) **7** (2008) 1–53
5. Nenova, S., Kästner, D.: Source Level Worst Case Timing Estimation and Architecture Exploration in Early Design Phases. In Holsti, N., ed.: Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009), Dublin, Ireland (2009) 12–22 (Preliminary proceedings).
6. AbsInt: aiT tool homepage (2008) `www.absint.com/ait`.
7. Scade: Homepage for Scade tool suite from Esterel (2009) `www.esterel-technologies.com/products/scade-suite`.
8. Souyris, J., Pavec, E.L., Himbert, G., Jegu, V., Borios, G., Heckmann, R.: Computing the worst case execution time of an avionics program by abstract interpretation. In Wilhelm, R., ed.: Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005). (2005)
9. Ascet: Homepage of ETAS' ascet tool chain (2009) `www.etas.com/en/products/ascet_software_products.php`.
10. Lisper, B., Santos, M.: Model identification for WCET analysis. In: Proc. 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09), San Francisco, CA, IEEE Computer Society (2009) 55–64
11. Stappert, F.: From Low-Level to Model-Based and Constructive Worst-Case Execution Time Analysis. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn (2004) C-LAB Publication, Vol. 17, Shaker Verlag, ISBN 3-8322-2637-0.
12. Erpendbach, E., Altenbernd, P.: Worst-case execution times and schedulability analysis of statecharts models. In: Proc. 11th Euromicro Conference of Real-Time Systems. (1999) 70–77
13. Kirner, R., Lang, R., Freiberger, G., Puschner, P.: Fully automatic worst-case execution time analysis for Matlab/Simulink models. In: Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02), Washington, DC, USA (2002)

14. Persson, P., Hedin, G.: Interactive execution time predictions using reference attributed grammars. In: Proc. of the 2:nd Workshop on Attribute Grammars and their Applications (WAGA'99), Netherlands. (1998) 173–184

15. Bernat, G., Burns, A., Wellings, A.: Portable Worst-Case Execution Time Analysis using Java Byte Code. In: Proc. 12th Euromicro Conference of Real-Time Systems, (ECRTS'00), Stockholm (2000) 81–88

16. Bate, I., Bernat, G., Murphy, G., Puschner, P.: Low-level Analysis of a Portable Java Byte Code WCET Analysis Framework. In: Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00). (2000) 39–48

17. Bartlett, M., Bate, I., Kazakov, D.: Guaranteed loop bound identification from program traces for WCET. In: Proc. 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09), San Francisco, CA, IEEE Computer Society (2009)

18. Rapitime: Rapitime white paper (2009)
    `www.rapitasystems.com/system/files/RapiTime-WhitePaper.pdf`.

19. Franke, B.: Fast cycle-approximate instruction set simulation. In Falk, H., ed.: Proc. 11th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2008), Munich (2008) 69–78

20. Chatterjee, S., Hadi, A.S.: Regression Analysis by Example. 4 edn. John Wiley & Sons (2000) ISBN 0-471-31946-5.

21. Mälardalen University: WCET project homepage (2009)
    `www.mrtc.mdh.se/projects/wcet`.

22. Gustafsson, J., Ermedahl, A., Lisper, B., Sandberg, C., Källberg, L.: ALF – a language for WCET flow analysis. In: Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009), Dublin, Ireland (2009) 1–11

23. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06). (2006)

24. Sandberg, C., Ermedahl, A., Gustafsson, J., Lisper, B.: Faster WCET flow analysis by program slicing. In: Proc. ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'06). (2006) 103–112

25. Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Rochange, C., ed.: Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007), Pisa, Italy (2007)

26. Ermedahl, A.: A Modular Tool Architecture for Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden (2003)