

A Review of Worst-Case Execution-Time Analysis (Editorial)

Peter Puschner

peter@vmars.tuwien.ac.at

*Institut für Technische Informatik, Technische Universität Wien, A-1040 Wien,
Austria*

Alan Burns

burns@minster.york.ac.uk

*Department of Computer Science, University of York, York, YO01 5DD, United
Kingdom*

September 24, 1999

A development process for safety-critical real-time computer systems has to emphasize the importance of time. On the one hand, such a development process has to be based on hardware and software technology that supports predictability in the time domain. On the other hand, the development process has to provide tools for assessing and verifying the correctness of the timing of both the hardware and the software components of the real-time systems being developed.

Together with schedulability analysis, *Worst-case execution time analysis (WCET analysis)* forms the basis for establishing confidence into the timely operation of a real-time system. WCET analysis does so by computing (upper) bounds for the execution times of the tasks in the system. These bounds are needed for allocating the correct CPU time to the tasks of an application. They form the inputs for schedulability tools, which test whether a given task set is schedulable (and will thus meet the timing requirements of the application) on a given target system.

While schedulability analysis is one of the traditional fields of investigation in real-time systems research, WCET analysis caught the attention of the research community only about ten years ago (Kligerman and Stoyenko, 1986; Mok et al., 1989; Puschner and Koza, 1989; Shaw, 1989). In the last decade, however, more and more research groups started to put a focus on WCET analysis. As a result, substantial progress has been made in this area in a relatively short time.

After ten years of research in the field, it is appropriate to have a special issue on WCET analysis. It is the goal of this special issue to review the achievements in WCET analysis and to report about the recent advances in this field. In the following section we will define the problem area of WCET analysis and thus clarify the issue WCET analysis is dealing with — still many people mix up execution-time analysis and response-time analysis. We will then summarize the subproblems



© 1999 Kluwer Academic Publishers. Printed in the Netherlands.

of WCET analysis and provide an overview of previous contributions to the state of the art in this field. At the end of the introduction we will give an overview to the research papers that have been selected for this special issue.

1. The Scope of Worst-Case Execution Time Analysis

The knowledge of the maximum time consumption of all pieces of code is a prerequisite for analyzing the worst-case timing behavior of a real-time system and for verifying its temporal correctness. This maximum time needed by each piece of code is assessed by means of WCET analysis. Informally, WCET analysis is defined as follows:

WCET analysis computes upper bounds for the execution times of pieces of code for a given application, where the *execution time of a piece of code* is defined as the time it takes the processor to execute that piece of code.

This informal definition of WCET analysis incorporates several points that are worth mentioning:

- WCET analysis computes *upper bounds* for the WCET, i.e., it does not guarantee to return the WCET exactly. Also, the definition does not make any statement about the quality of the results of WCET analysis.
- The WCET bound computed for a piece of code is *application-dependent*, i.e., a single piece of code may have different WCETs, and thus WCET bounds in different application contexts.
- WCET analysis assesses the duration that the processor is *actually executing* the analyzed piece of code, i.e., assuming non-preemptive execution. It is important to note that the results of WCET analysis do not include waiting times due to preemption, blocking, or other interference.
- WCET analysis is *hardware-dependent*. WCET analysis therefore has to model the features of the target hardware on which the code is supposed to execute.

1.1. GOALS OF WCET ANALYSIS

The WCET bounds resulting from WCET analysis determine the time quanta that a system designer or a scheduling tool must reserve for

the execution of tasks. In order to make real-time systems temporally predictable and to keep the price of such systems reasonable, WCET analysis must fulfill the following requirements:

- *Provision of safe bounds.* The results of WCET analysis are used for the validation of the temporal correctness of real-time systems. The computed execution time bounds thus have to be safe, i.e., they must not under-estimate the worst case.
- *Tightness of computed WCET bounds.* The bounds on the WCETs of tasks determine the allocation of CPU time. The more pessimistic the result of WCET analysis, the more CPU time must be reserved for each task and the higher is the chance that the available resources are incorrectly judged as being insufficient for the given set of tasks. High-quality WCET analysis avoids such pessimistic results and the costs associated with the compensation for this pessimism (e.g., hand-crafted modifications to the assembler or machine code, additional or faster hardware).

1.2. PROBLEMS OF WCET ANALYSIS

This section identifies the three major problem domains that have to be addressed to meet the goals of WCET analysis. These problem domains are derived from the information that is needed for computing the worst-case execution time of a program. The WCET of a program in a specific application context is determined by

- the possible sequences of program actions in the application and
- the time needed for each action in each of these sequences.

This statement is independent of the level of code representation, i.e., it holds for the source-code representation as well as for the machine-language representation of a piece of program code.

Many of today's approaches to WCET analysis demand that the programmer provide information about (in)feasible execution paths of the code to be analyzed. This path information is described at the high-level language interface. On the other hand, the actual computation of WCETs, that uses this path information, takes place at the machine-language level, where the execution times of basic actions can be accurately modeled. Today's practical approaches to WCET analysis therefore have to bridge the gap between the two different representation levels. They have to translate the path information provided at the source level to the machine-language representation of a program.

Figure 1 illustrates the three problem domains of WCET analysis. The *characterization of execution paths* takes place on the source-language level (upper box). In order to be analyzed for its WCET, the source code and the path information are translated to the machine-language representation (middle box). The adequate *translation of path information* is part of WCET analysis. Finally, the *hardware-level execution-time analysis* investigates the execution times of the possible program paths and computes a WCET bound (lower box). In the following the three problem domains are described in more detail.

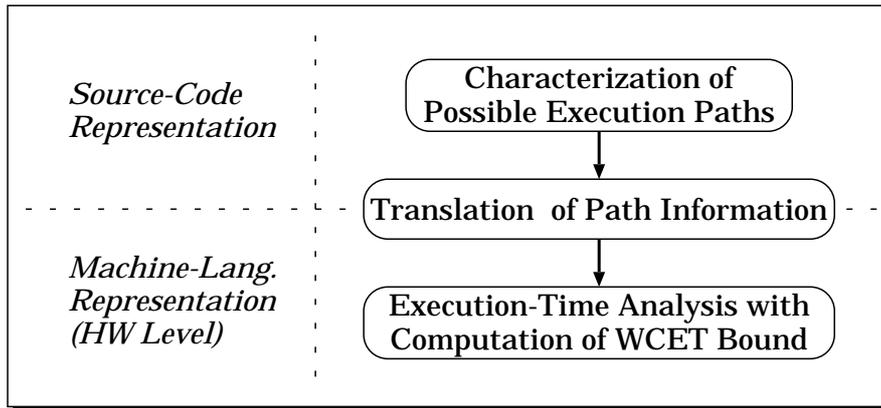


Figure 1. Problem Domains of WCET Analysis.

1.2.1. Characterization of Execution Paths

The potential sequences of actions during program execution are described by the source code of a program. The control semantics of the constructs of a real-time programming language¹ define the possible execution sequences of statements. As mentioned above, this information is in general not sufficient to bound the WCET of a program automatically.

Researchers working on the *characterization of execution paths* investigate methods for describing path information for WCET analysis. They search for program annotation methods and constructs that are adequate for describing execution frequencies and execution sequences of program parts and their dependencies (see, e.g., (Kligerman and Stoyenko, 1986; Puschner and Koza, 1989; Park, 1993)). More recent work uses semantic analysis to support the programmer in path analysis, e.g., (Altenbernd, 1996).

¹ We only focus on imperative languages, since most real-time programming languages are imperative.

1.2.2. *Translation between Source-Language and Machine-Language Representation*

Translating the path information specified at the source-language level to the machine-language representation is not trivial. Compilers, especially optimizing compilers, re-arrange code and perform transformations to obtain efficient code and reach their optimization goals (high execution speed, low memory consumption, etc.). These changes in the code structure make it difficult to identify the execution paths that have been characterized at the source level in the machine code.

Researchers are searching for different ways to bridge the gap between the two different code representations. One method is to let the compiler transform the path information in parallel with code translation. This approach was followed in (Pospischil et al., 1992; Vrchaticky, 1994; Puschner, 1998a). Adapting or writing a compiler is unacceptable or impossible in some cases (high costs, sources are not available, etc.). Therefore approaches are sought which are either compiler-independent (Puschner, 1998b) or which need only limited compiler assistance (Engblom et al., 1998). A completely different option, that avoids the problem of describing and translating path information, is to try to automatically derive path information from the machine or assembly code. Since the problem of automated path analysis is in general undecidable, the applicability of this strategy is restricted to “good-natured” programs.

1.2.3. *Hardware-Level Execution-Time Analysis*

WCET analysis needs the execution time of each program action on each possible program path. Since the knowledge about execution times of program actions is most accurate at the machine-code level, this level is the preferred choice for modeling the execution times of program actions: a program action corresponds to a machine instruction or to a sequential block of machine instructions.

The aim of hardware-level execution-time analysis is to derive the duration that each instance of an instruction needs for its execution on a given hardware and to compute the WCET bound for a given piece of code from the execution times of the single instructions. As mentioned above, this analysis has to assess the possible execution sequences of instructions and the timing of each instruction in each of the possible sequences. The complexity of this analysis depends on the features of the target hardware. In many modern architectures the execution time of an instruction not only depends on the operation being performed and its operands but also on the hardware state (contents of caches, pipelines, etc.) at the time the instruction executes. Recent research focusing on hardware-level execution-time analysis can be found in

(Zhang et al., 1993; Healy et al., 1995; Hur et al., 1995; Kim et al., 1996; Li et al., 1995; Li et al., 1996; Mueller, 1997; Lim et al., 1998).

The preceding sections showed that WCET analysis has to deal with a set of complex problems — the characterization and the analysis of execution paths, the translation of path information from the source code to the machine code of a program, and the hardware-level WCET computation. The quality of the solutions found in each of these problem areas determines the value of the entire WCET analysis. Only by providing adequate solutions to all three problems can one achieve the goals of WCET analysis: the safeness and the tightness of WCET bounds.

2. Contributions of Different Institutions

REAL-TIME EUCLID, UNIVERSITY OF TORONTO

One of the first papers on WCET analysis was written by Kligerman and Stoyenko. In (Kligerman and Stoyenko, 1986; Stoyenko, 1987), they discussed restrictions for programming languages that are necessary to allow for a computation of execution time bounds for real-time tasks. Their programming language, called Real-Time Euclid, prohibits the use of recursions and goto-statements. Loops are restricted to *time bounded loops* and simple *for*-loops. For the latter the maximum number of iterations, and thus the time maximally spent in these loops, can easily be derived. An algorithm that calculates an execution time bound of a RT-Euclid program by computing the maximum duration of all possible communication and execution sequences of the code pieces of the program is described in detail in (Stoyenko et al., 1991). In principle, the proposed algorithm unrolls all loops of a program and then searches for the execution sequence with a maximum duration.

TIMING SCHEMA, UNIVERSITY OF WASHINGTON

A more formal approach for the computation of worst-case execution times is described by Park and Shaw (Shaw, 1989; Park and Shaw, 1991). The timing schemes described allow Park and Shaw to compute the minimum and maximum execution times of constructs common in most programming languages. Based on this theory a timing tool was implemented. This tool computes execution time bounds by analyzing the source code of C programs. It interacts with the user to obtain loop bounds required for the calculations. When the timing tool was evaluated the computed execution time bounds were very close to the real bounds (Park and Shaw, 1991).

In (Park, 1992; Park, 1993), Park extended the previous work. Regular expressions are used to characterize feasible paths, thus improving the quality of the computed execution time bounds. When the analyzer tool computes execution time bounds it takes the maximum of the execution time bounds computed for all possible path groups.

TIMING SCHEMA, SEOUL NATIONAL UNIVERSITY

Shaw's timing-schema approach has been taken up by the real-time systems group at Seoul National University. In a first step, the approach was modified to work for pipelined processor architectures. In (Lim et al., 1994) the authors introduce so-called WCTAs (worst-case timing abstractions), sets of tables that describe the reservation of the different units of a CPU at the beginning and the end of basic blocks of assembler programs. They adapt the timing-schema operations to work on WCTAs instead of execution time values, thus achieving up to 50% improvements of computed WCET bounds.

The WCTA approach was later modified to take into account caching effects. The work (Hur et al., 1995) describes how direct mapped instruction caches are modeled in timing schemes. Experiments compare the timing schema that do not take into account caches with timing schema that take into account caches. In the results the bounds that do not take into account the cache are in general 100% higher than the bounds that were computed with consideration of the cache.

Further improvements of the timing-schema approach added mechanisms to model data caching (Kim et al., 1996) and the timing of simple multi-issue machines (Lim et al., 1998). The authors of (Kim et al., 1996) aim at reducing the pessimism of WCET bounds due to dynamic data load and store instructions. In a first step they apply global data flow analysis to avoid data accesses being mis-classified as dynamic. The data-dependence analysis applied as a second step tries to minimize the adverse effects of dynamic load and stores by taking into account control-flow information. The presented work reports about 20% improvements of computed WCET bounds with this method for programs with a large number of dynamic load and store instructions.

TIMETOOL, UNIVERSITY OF TEXAS AT AUSTIN

Mok and his group (Mok et al., 1989; Amerasinghe, 1985; Chen, 1987) produced a timing tool for assembly language programs, which are annotated by TAL (timing analysis language) scripts. TAL scripts are written by the programmer. They describe timing properties of code pieces that cannot be automatically derived from the code, like loop bounds and additional control flow information. The timetool uses the

TAL scripts together with the assembly language code to compute WCET bounds of reasonable quality. A drawback of this approach is that it does not provide a high-level language interface. The use of the tool is restricted to the analysis of assembly language programs.

VIENNA UNIVERSITY OF TECHNOLOGY (TU VIENNA)

At the Vienna University of Technology, researchers of two groups — the Real-Time Systems Group and the Automation Systems Group — are working on WCET analysis. Work on WCET analysis of the *Real-Time Systems Group* was first reported in (Puschner and Koza, 1989). Puschner and Koza analyzed high level language code to compute bounds for the execution times of tasks by extending the bounded loop concept found in (Kligerman and Stoyenko, 1986). New language constructs describe additional knowledge about the control flow. This allows the developers of programs to derive significantly tighter execution time bounds for more complex programs. The described tool combines the source program plus the compiled code to compute the WCET of a program. The problem of this tool approach is that such a tool does not guarantee to find the correct mapping between the two representations for all input programs.

To avoid this problem, a new approach that combines compilation and WCET analysis was developed (Puschner and Schedl, 1993). The compiler, WCET analyzer, and an intelligent editor were integrated into a special real-time programming environment (Pospischil et al., 1992). In this environment the compiler produces the input for the WCET analyzer: the *timing tree*, a high-level language description of a program's syntactic structure, execution constraints expressed by the user, and the execution times of the basic code constituents (Vrchoťický, 1994), where they assume a simple hardware architecture without caches. The editor provides features to display execution time information for every programming language construct at the source code level.

The WCET analyzer for the timing tree uses a graph-based WCET computation method. It computes WCET bounds with an ILP (integer linear programming) solver (Puschner and Schedl, 1997; Puschner, 1998a). The WCET tool not only allows its users to compute worst-case execution time bounds of high quality; it also produces detailed information about the contribution of every statement to this bound and allows programmers to experiment with hypothetical times.

In parallel to the development of WCET computation concepts and techniques, the potential benefits of using user-supplied path information were investigated. Despite the variety of path-description mecha-

nisms used in literature, only (Puschner and Vrchoticky, 1991) assessed the possible benefits of these mechanisms. The experiments reported in this work demonstrated that user-supplied path information plays an important role for computing WCET bounds of good quality.

In the *Automation Systems Group* Blieberger and Lieger realized so-called discrete loop constructs for real-time programming languages to simplify the computation of upper iteration bounds (Blieberger, 1994). Discrete loop constructs demand from the programmer that he/she describes in which way each loop-invariant variable changes from one iteration to the next iteration. This information is then used by the timing analysis tool to draw conclusions on how often each loop may iterate in the worst case. Further work by Blieberger and Lieger (Blieberger and Lieger, 1996) describes a method to compute both the WCET and the maximum space consumption of recursive procedures.

UNIVERSITY OF MASSACHUSETTS AT AMHERST

Niehaus' work was conducted as part of the Spring project (Stankovic and Ramamritham, 1991). In (Niehaus et al., 1991; Niehaus, 1991), he analyses intermediate code to compute execution time bounds of non-preemptable program segments on computer architectures with instruction caches. In order to provide a consistent state at the beginning of the execution of each such program segment he proposes to flush caches whenever a context switch occurs. Starting from this predictable state he analyses the speedup of program segments due to instruction caching. The analysis method that is sketched in this work considers the speedup due to spatial locality of instruction sequences and the temporal locality of very simple loops.

UNIVERSITY OF YORK

Work at York has focused on two distinct topics; high-level path analysis and low-level pipeline prediction.

Chapman (Chapman et al., 1994; Chapman et al., 1996) based his analysis on the SPARK language. SPARK is based on a subset of Ada 83 but includes annotations for static analysis. For example, procedures may have pre- and post-conditions specified. Flow analysis is supported by tools. In particular the *verification condition*(VC) of a path may be handled by a theorem prover.

Chapman applied the techniques to a number of industrial case studies. In all of these examples the degree of pessimism was significantly reduced – in one case from 1630% to 37%. The remaining pessimism was due to using a simple low-level model. He also showed how it was

possible to analyze exception propagation flow and to include exception handlers in basic paths (Chapman et al., 1993).

An early attempt to model pipelines in a worst-case execution time tool was performed by Zhang and Burns (Zhang et al., 1993). They considered an Intel 80C188 processor which consists of a two stage pipeline with the first stage being a simple instruction prefetch. Although the pipeline is simple, much of the complication in the analysis comes from trying to determine how much of the next instruction to be executed is in the prefetch queue. However, the results reported indicate that the degree of pessimism in WCET of a typical basic block can be reduced from over 20% (if no pipeline is assumed) to 2% with the pipeline analysis.

PRINCETON UNIVERSITY

Li and Malik (Li and Malik, 1995) developed a method for computing WCET bounds of programs on processor architectures with pipelines and caches. Worst case program execution times are computed with an integer linear programming (ILP) solver. Disjoint sets of program paths are identified and a linear programming problem is constructed for each path set. Then, the solution of each programming problem is computed. The maximum value of the solutions is the worst-case execution time. While earlier work of Li, Malik, and Wolfe limits its focus on computer architectures with pipelines and direct-mapped instruction caches (Li et al., 1995), later findings make the technique applicable to set-associative data and instruction caches with other line-replacement strategies (Li et al., 1996).

FLORIDA UNIVERSITIES AND HUMBOLDT UNIVERSITY BERLIN

One focus of research conducted at these institutions is on the modeling of execution times for modern RISC processor architectures. Harmon introduced a WCET analysis technique called micro-analysis to analyze the execution times of machine instructions at the micro-instruction level (Harmon, 1991; Harmon et al., 1992). This fine-grained modeling made possible to improve the results of machine-instruction based WCET analysis.

A number of articles introduce and refine an approach for bounding the worst-case cache performance of programs. In (Healy et al., 1999) static cache simulation for instruction caches is introduced. Static cache simulation analyses the possible control flows of programs and statically categorizes the caching behavior of each access to the instruction memory (The categorizations are “always hit”, “always miss”, “first hit”, “first miss”). The timing analyzer uses the categorization information

to compute execution time bounds. This is performed for each loop level of nested loops and for each calling instance of each function in its possible calling sequences, and finally for entire tasks.

The work described in (Healy et al., 1999) found its continuation in different ways. First, this work was extended by pipeline analysis (Healy et al., 1995). Another development is (Mueller, 1997). In this work Mueller generalizes the static cache simulation of direct-mapped caches to set-associative caches. He computes WCET bounds for several programs with this method and shows that the quality of static cache analysis for set-associative caches is as good as for direct-mapped caches.

In parallel to the work on hardware-level WCET analysis, a programming environment for hard-real time systems programming was developed. This environment allows users to specify the timing constraints for programs in the program code and displays worst-case path and timing information both on the source language level and the assembly language level of the program code (Ko et al., 1996). While this environment requires some user interaction to specify loop bounds or time limits for code execution, further work aims at automating this step (Healy et al., 1998). The presented method tries to derive the minimum and the maximum number of loop iterations by analyzing the program source code. The approach is capable of dealing with simple dependencies of loop bounds of nested loops.

C-LAB AND UPPSALA UNIVERSITY

Altenbernd's (Altenbernd, 1996) work on path analysis applies symbolic execution to program code in order to identify false paths, i.e., execution paths which, due to data dependencies, can never be executed. The goal of this work is to derive a WCET calculation approach which needs no more path information from the user than the maximum loop counts for all loops and the maximum depths of all recursions.

The work presented in (Stappert, 1997) describes a solution for the instruction-level timing analysis of programs. The proposed solution considers processor features like pipelines, data caches, and instruction caches. It is, however, very restrictive in that it assumes the analyzed code to be synthesized code without loops, without procedure calls, and with access to static variables only. The described tool makes use of data-flow analysis to limit the pessimism of the WCET estimates. In the experiments reported by the author the pessimism of the WCET bounds was of the order of 10%.

Further work (Engblom et al., 1998) describes a technique to map path information from the source code to an object-code representa-

tion. The authors call this technique co-transformation. In order to cope with structural changes in the program due to code optimization, the compiler generates a log of its transformations. The log is coded in a special-purpose optimization description language (ODL) that is interpreted by the WCET analyzer to map the path information from the source code to the re-structured object code. An example illustrates how WCET estimates are then computed using the transformed path information.

UNIVERSITY OF THE SAARLAND

The researchers in Reinhard Wilhelm's group investigate the possibility to structure WCET analysis into distinct phases. In their approach (Theiling and Ferdinand, 1998) each of the two phases focuses on a dedicated task and each phase applies an adequate method to solve its task. The first phase assesses the timing of instructions on the target hardware. It uses abstract interpretation to perform the cache analysis and the pipeline analysis (Alt et al., 1996). The second phase, called path analysis, uses the results from the cache and pipeline analysis and forms an ILP problem for the computation of WCET bounds. The evaluation of the ILP problem yields the WCET analysis results.

Wilhelm and his group built a prototype tool for their analysis approach and evaluated their two-phased strategy. They report that in the experiments performed the tool computed reasonable WCET bounds within just a few seconds (Theiling and Ferdinand, 1998).

3. The Articles of this Issue

The six papers that were selected for this issue present solutions to the different problem domains of WCET analysis.

The first paper, by Healy, Sjodin, Rustagi, Whalley, and van Engelen, entitled "Supporting Timing Analysis by Automatic Bounding of Loop Iterations", describes the automatic determination of loop bounds for a large class of loops typically appearing in real-time software. The information about bounds is used in the formation of symbolic summation expressions to derive accurate bounds for non-rectangular and data-dependent loops.

In the second paper, "Fast and Precise Worst-Case Execution-Time Prediction by Separated Cache and Path Analysis", Theiling, Ferdinand, and Wilhelm propose a divide-and-conquer approach to deal with both path and hardware analysis. In the proposed approach they first model the timing of the cache using abstract interpretation. In the

subsequent path analysis they construct an integer linear programming (ILP) problem whose solution yields the WCET bound.

The third contribution to the issue, “Symbolic Cache Analysis for Real-Time Systems”, is authored by Blieberger, Fahringer, and Scholz. The authors describe a two-step process for cache analysis. In the first step a symbolic evaluation of a C program produces a so-called symbolic tracefile, a file that characterizes the dynamic behavior of the program. The second step of the analysis evaluates the tracefile to predict cache hits and misses and computes WCET bounds from this information.

The fourth paper, by Mueller, is titled “Timing Analysis for Instruction Caches”. Mueller proposes to analyze the possible sequencing of instruction and to categorize instructions according to possible cache hits and misses. The actual computation of WCET bounds is then based on the categorization of the instructions and the information about the timing parameters of the hardware.

While the previous papers model the timing of caches, the last two papers of this issue try to include the timing effects of other, processor-specific hardware features in WCET analysis.

The fifth paper, by Colin and Puaut, “Worst Case Execution Time Analysis for a Processor with Branch Prediction”, bounds the pessimism in hardware analysis that is due to erroneous branch prediction. The authors build a categorization of branch instructions in a machine program that reflects the possibilities that the branch-prediction hardware will predict the control flow correctly or incorrectly. Based on this categorization the possible program paths are analyzed for their timing and WCET bounds are computed.

The final paper focuses on the WCET analysis for super-scalar processors. This paper, by F. Burns, Koelmans, and Yakovlev is entitled “WCET Analysis of Super-scalar Processors using Simulation with Coloured Petri Nets”. In a simulation-based approach the authors use colored petri nets to evaluate the timing of instruction execution on modern processors. Their petri nets model the timing of speculative and out-of-order instruction execution for processors with multiple execution units and with multi-staged pipelines.

We feel that this first special issue of a journal on WCET analysis gives the reader a good overview of the current state of the art in this field, although the selected papers do not – indeed they cannot – cover the entire work done in the area. The issue offers (different) solutions to some of the problems involved in bounding a program’s WCET and contains an extensive list of references to work on other directly related issues. It thus forms a comprehensive source of information for all those people who need to analyze the WCET of a program or who want to further develop the state of the art in this field.

References

- Alt, M., C. Ferdinand, F. Martin, and R. Wilhelm: 1996, 'Cache Behavior Prediction by Abstract Interpretation'. In: *Proc. of the Static Analysis Symposium, LNCS 1145*. pp. 52–66.
- Altenbernd, P.: 1996, 'On the False Path Problem in Hard Real-Time Programs'. In: *Proc. Euromicro Workshop on Real-Time Systems*. L'Aquila, Italy.
- Amerasinghe, P.: 1985, *A Universal Hardware Simulator*. Dept. of Computer Sciences, University of Texas, Austin, TX, USA: Undergraduate Honors Thesis.
- Blieberger, J.: 1994, 'Discrete Loops and Worst Case Performance'. *Computer Languages* **20**(3), 193–212.
- Blieberger, J. and R. Lieger: 1996, 'Worst-Case Space and Time Complexity of Recursive Procedures'. *Real-Time Systems* **11**(2), 115–144.
- Chapman, R., A. Burns, and A. Wellings: 1996, 'Combining Static Worst-Case Timing Analysis and Program Proof'. *Real-Time Systems* **11**(2), 145–171.
- Chapman, R., A. Burns, and A. J. Wellings: 1993, 'Worst-case timing analysis of exception handling in Ada'. In: L. Collingbourne (ed.): *Ada: Towards Maturity. Proceedings of the 1993 AdaUK conference*. pp. 148–164.
- Chapman, R., A. Burns, and A. J. Wellings: 1994, 'Integrated program proof and worst-case timing analysis of SPARK Ada'. In: *ACM Workshop on language, compiler and tool support for real-time systems*.
- Chen, M.: 1987, *A Timing Analysis Language - (TAL)*. Dept. of Computer Sciences, University of Texas, Austin, TX, USA: Programmer's Manual.
- Engblom, J., A. Ermedahl, and P. Altenbernd: 1998, 'Facilitating Worst-Case Execution Times Analysis for Optimized Code'. In: *Proc. Euromicro Workshop on Real-Time Systems*. Berlin, Germany, pp. 146–153.
- Harmon, M. G.: 1991, 'Predicting Execution Time on Contemporary Computer Architectures'. Ph.D. thesis, Department of Computer Science, Florida State University, Tallahassee, FL, USA.
- Harmon, M. G., T. P. Baker, and D. B. Whalley: 1992, 'A Retargetable Technique for Predicting Execution Time'. In: *Proc. 13th Real-Time Systems Symposium*. Phoenix, AZ, USA, pp. 68–77.
- Healy, C., M. Sjördin, V. Rustagi, and D. Whalley: 1998, 'Bounding Loop Iterations for Timing Analysis'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. Denver, CO, pp. 12–21.
- Healy, C. A., R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon: 1999, 'Bounding Pipeline and Instruction Cache Performance'. *IEEE Transaction on Computers* **48**(1), 53–70.
- Healy, C. A., D. B. Whalley, and M. G. Harmon: 1995, 'Integrating the Timing Analysis of Pipelining and Instruction Caching'. In: *Proc. 16th Real-Time Systems Symposium*. pp. 288–297.
- Hur, Y., Y. H. Bae, S. Kin, B. Rhee, W. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim: 1995, 'Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study'. In: *Proc. 16th Real-Time Systems Symposium*. pp. 308–319.
- Kim, S., S. L. Min, and R. Ha: 1996, 'Efficient Worst Case Timing Analysis of Data Caching'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. pp. 230–240.
- Kligerman, E. and A. Stoyenko: 1986, 'Real-Time Euclid: A Language for Reliable Real-Time Systems'. *IEEE Transactions on Software Engineering* **SE-12**(9), 941–949.

- Ko, L., C. Healy, E. Ratliff, and M. Harmon: 1996, 'Supporting the Specification and Analysis of Timing Constraints'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. pp. 170–178.
- Li, Y. S. and S. Malik: 1995, 'Performance Analysis of Embedded Software Using Implicit Path Enumeration'. In: *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*. La Jolla, CA, USA, pp. 95–105.
- Li, Y. S., S. Malik, and A. Wolfe: 1995, 'Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software'. In: *Proc. 16th Real-Time Systems Symposium*. pp. 298–307.
- Li, Y. S., S. Malik, and A. Wolfe: 1996, 'Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches'. In: *Proc. 17th Real-Time Systems Symposium*.
- Lim, S., Y. H. Bea, G. T. Jang, B. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim: 1994, 'An Accurate Worst Case Timing Analysis for RISC Processors'. In: *Proc. 15th Real-Time Systems Symposium*. pp. 97–108.
- Lim, S., J. H. Han, J. Kim, and S. L. Min: 1998, 'A Worst Case Timing Analysis Technique for Multiple-Issue Machines'. In: *Proc. 19th Real-Time Systems Symposium*. pp. 334–345.
- Mok, A. K., P. Amerasinghe, M. Chen, and K. Tantisirivat: 1989, 'Evaluating Tight Execution Time Bounds of Programs by Annotations'. In: *Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software*. Pittsburgh, PA, USA, pp. 74–80.
- Mueller, F.: 1997, 'Generalizing Timing Predictions to Set-Associative Caches'. In: *Proc. Euromicro Workshop on Real-Time Systems*. Toledo, Spain, pp. 64–71.
- Niehaus, D.: 1991, 'Program Representation and Translation for Predictable Real-Time Systems'. In: *Proc. 12th Real-Time Systems Symposium*. pp. 53–63.
- Niehaus, D., E. Nahum, and J. A. Stankovic: 1991, 'Predictable Real-Time Caching in the Spring System'. In: *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*. Atlanta, GA, USA, pp. 80–87.
- Park, C. Y.: 1992, 'Predicting Deterministic Execution Time of Real-Time Programs'. Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA.
- Park, C. Y.: 1993, 'Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths'. *Real-Time Systems* **5**(1), 31–62.
- Park, C. Y. and A. C. Shaw: 1991, 'Experiments with a Program Timing Tool Based on Source-Level Timing Schema'. *IEEE Computer* **24**(5), 48–57.
- Pospischil, G., P. Puschner, A. Vrchoticky, and R. Zainlinger: 1992, 'Developing Real-Time Tasks with Predictable Timing'. *IEEE Software* **9**(5), 35–44.
- Puschner, P.: 1998a, 'A Tool for High-Level Language Analysis of Worst-Case Execution Times'. In: *Proc. Euromicro Workshop on Real-Time Systems*. Berlin, Germany, pp. 130–137.
- Puschner, P.: 1998b, 'Worst-Case Execution Time Analysis at Low Cost'. *Control Engineering Practice* **6**(1), 129–135.
- Puschner, P. and C. Koza: 1989, 'Calculating the Maximum Execution Time of Real-Time Programs'. *Real-Time Systems* **1**(2), 159–176.
- Puschner, P. and A. Schedl: 1993, 'A Tool for the Computation of Worst Case Task Execution Times'. In: *Proc. Euromicro Workshop on Real-Time Systems*. Oulu, Finland, pp. 224–229.
- Puschner, P. and A. Schedl: 1997, 'Computing Maximum Task Execution Times — A Graph-Based Approach'. *Real-Time Systems* **13**(1), 67–91.

- Puschner, P. and A. Vrchoticky: 1991, 'An Assessment of Task Execution Time Analysis'. In: *Proc. 10th IFAC Workshop on Distributed Computer Control Systems*. Semmering, Austria, pp. 41–45. published by Pergamon Press, Oxford, New York, 1992.
- Shaw, A. C.: 1989, 'Reasoning About Time in Higher-Level Language Software'. *IEEE Transactions on Software Engineering* **SE-15**(7), 875–889.
- Stankovic, J. A. and K. Ramamritham: 1991, 'The Spring Kernel: A New Paradigm for Real-Time Operating Systems'. *IEEE Software* pp. 62–72.
- Stappert, F.: 1997, 'Predicting Pipelining and Caching Behaviour of Hard Real-Time Programs'. In: *Proc. Euromicro Workshop on Real-Time Systems*. Toledo, Spain, pp. 80–86.
- Stoyenko, A.: 1987, *A Real-Time Language With A Schedulability Analyzer*. Computer Systems Research Institute, University of Toronto, Canada: Dissertation.
- Stoyenko, A., V. Hamacher, and R. Holt: 1991, 'Analyzing Hard-Real-Time Programs for Guaranteed Schedulability'. *IEEE Transactions on Software Engineering* **SE-17**(8), 737–750.
- Theiling, H. and C. Ferdinand: 1998, 'Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis'. In: *Proc. 19th Real-Time Systems Symposium*. Madrid, Spain, pp. 144–153.
- Vrchoticky, A.: 1994, 'Compilation Support for Fine-Grained Execution Time Analysis'. In: *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*. Orlando FL.
- Zhang, N., A. Burns, and M. Nicholson: 1993, 'Pipelined Processors and Worst Case Execution Times'. *Real-Time Systems* **5**(4), 319–343.