Worst-Case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System

Martin Carlsson[1], Jakob Engblom[2], Andreas Ermedahl[3], Jan Lindblad[1], and Björn Lisper[4]

[1] Enea OSE Systems AB, P.O. Box 232, SE-183 23 Täby, SWEDEN
carlssonm@chello.se, janl@enea.se
[2] IAR Systems AB, P.O. Box 23051, SE-750 32 Uppsala, SWEDEN
jakob.engblom@iar.se
[3] IT-Dept., Uppsala University, P.O. Box 337, SE-751 05 Uppsala, SWEDEN
andreas.ermedahl@it.uu.se
[4] Dept. of Computer Engineering, Mälardalen University, P.O. Box 883, SE-721 23 Västerås, SWEDEN
bjorn.lisper@mdh.se

Abstract. Worst-Case Execution Time (WCET) analysis has been around for some time now, but has so far not been much used to analyse real production codes. Here, we present a case study where static WCET analysis was used to find upper time bounds for time-critical regions in a commercial real-time operating system. We report on practical experiences from the work, like the reverse engineering required to find these regions and prepare them for the analysis. We give the results of the WCET analysis and discuss the precision. We also present some qualitative and quantitative data on the program structure of the regions. This information is useful to judge whether WCET analysis could provide any useful results for this class of real codes, without excessive manual labour. Finally, we present a "wishlist" for features of WCET analysis tools, which has emerged during the project, and comment on the feasibility of implementing these features.

1 Introduction

Real-time systems have timing requirements. Thess requirements imply upper limits on the execution times of codes in the systems: in order to guarantee that these requirements are met, it must be verified that the codes execute within their time limits. The purpose of *Worst-Case Execution Time* (WCET) analysis is to find upper bounds for the execution times of codes on certain processors. Apparently, some form of WCET analysis is needed to ensure the formal correctness of real-time systems. It is thus important to develop and evaluate such techniques.

Real-time systems most often use real-time operating systems. Certain parts of the operating system code will then be time-critical. An example is *disable interrupt regions* (or DI regions, for short), which execute with the interrupts turned off. A DI region can for instance be a critical section, where some shared resource is accessed.

The execution of DI regions can potentially delay any other activity in the system. It is therefore important to establish bounds on their execution times.

In this case study, a prototype tool for static WCET analysis was used to find upper bounds to the execution time for a number of DI regions in the delta kernel of the Enea OSE operating system [8]. The case study was done as an M. Sc. thesis project [2].

The purpose of the study was at least twofold. For the operating systems vendor, it is interesting to find DI regions that may have long execution times, so they can be optimized. This will improve the responsiveness of the operating system for soft real-time applications. Furthermore, guaranteed upper bounds on the execution times of these regions provide an increased confidence in the hard real-time properties of the operating system. Testing, which is the current method to validate real-time properties, does not provide any strict guarantees. Furthermore, testing the individual DI regions is hard. Measuring their execution times in isolation would require a costly instrumentation of the operating system code, with possible probing effects that could render the results useless anyway. Therefore, it is interesting to investigate whether WCET analysis can provide a feasible way to bound the execution times of the regions at a reasonable cost.

For the WCET analysis research, it is very important to be able to test analysis techniques on real programs. Toy benchmarks may fail to uncover the true difficulties. Time-critical parts of real-time operating systems is an interesting class of target codes for WCET analysis. Case studies like this can provide valuable information about the properties of such codes, that can be fed back to tune the analysis to handle this class of codes better.

Finally note that other formal techniques for verifying real-time properties will typically depend on the correctness of given WCET bounds for time-critical activities, so the absence of a formal WCET analysis reduces their applicability.

The rest of this paper is organized as follows. In Section 2, we give a brief introduction to WCET analysis and related work in the area. In Section 3 our current prototype WCET tool is described. Section 4 gives a short description of the OSE operating system. Sections 5 and 6 describe the chosen target processor for the analysis, the experimental setup, and the reverse engineering required to find the DI regions. In Section 7 we give the results, both the WCET estimates and facts about the structure of the analyzed code. Finally, in Section 8, we draw some conclusions and give ideas for further research.

2 WCET Analysis and Related Work

WCET analysis is usually divided into three parts: a fairly machine-independent *flow analysis* of the code, where information about the possible program flow paths is derived, a *low-level analysis* where the execution time for atomic parts of the code is decided from a performance model for the target architecture, and a final *calculation* where the information from these analyses is put together in order to derive the actual WCET bounds. The flow analysis is usually called “high-level analysis”, since it is often done on the source code, but it can equally well be done on intermediate or machine code level.

The flow analysis can be done by hand. The information must then be communicated through annotations, either integrated in the code [29] or provided separately [14, 17, 19, 30]. However, to some extent the flow analysis can be automated [4, 13, 15–17, 23, 33].

The low-level analysis can be further divided into two stages. *Global* low-level analysis takes features requiring a global view (caches, branch predictors) into account [5, 14, 16, 18, 20, 21, 24, 25, 33, 36]. *Local* low-level analysis considers features with local effects, like pipelines and superscalar instruction execution [9, 10, 16, 21, 22, 32, 33].

Three classes of calculation methods are mainly used: *tree-based* calculation, *path-based* calculation, and the *Implicit Path Enumeration Technique* (IPET). The tree-based approach is limited to well-structured codes, and assumes that the execution time bounds for programs can be directly derived from time bounds on their parts through simple rules [3, 28]. Path-based techniques explicitly explore the execution paths of a program fragment [16, 33, 34]. IPET, finally, models possible program flows with arithmetic constraints [11, 14, 17, 19, 27, 30]. IPET is formulated for (possibly unstructured) program flow graphs, with basic blocks connected by edges. Each entity i in the graph is given an execution time t_i by the low-level analysis, and an execution count x_i. The WCET is estimated by $\max(\sum_i x_i t_i)$ subject to the constraints given by the flow analysis. If these constraints are linear, then the optimization problem can be solved by integer linear programming (ILP) techniques. IPET can handle complex flow constraints, but this may come at the cost of an expensive calculation.

The only other work we know, where WCET techniques have been used to analyse operating system kernels, is by Colin and Puaut [6]. They analyse some operating system functions of RTEMS, a small, open-source real-time kernel. Their WCET analysis tool is based on the tree-based approach, which requires manual annotations to bound the number of loop iterations, and well-structured code. Some unstructured parts of RTEMS had to be rewritten in order to fit the tool.

We extract control flow graphs from object code for WCET analysis. This has also been done by Theiling [35].

3 The WCET Tool

Our WCET analysis tool targets embedded systems. The ultimate goal is to be able to analyse optimized code for such systems, generated by a compiler for full C, with reasonable precision for an interesting range of real embedded programs. For an overview, see [12].

The tool consists of an automatic flow analyser, currently under construction, a low level analysis, that basically estimates execution times for basic blocks, and a calculation that can be either IPET- or path-based. For the case study, the tool was extended with a “frontend” to extract DI regions from object code, and build control flow graphs for them. The current architecture, without the automatic flow analysis, is shown in Fig. 1.

The flow analysis will work on an intermediate format for an optimizing C compiler, that is currently under development [31]. This format is close to a control flow graph, and will accurately describe the control structure of the generated binary code. The

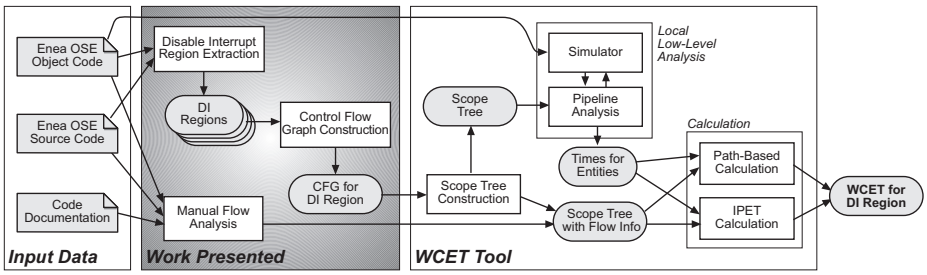

Fig. 1. Architecture of our WCET analysis tool.

reason to perform the flow analysis on this format, rather than on the source code, is that optimisations that change the program flow then are taken into account. The control flow graphs may correspond to unstructured programs, which excludes some methods to estimate the program flow. However, classical abstract interpretation [7] handles general control flow graphs, and our analysis will be based on an abstract interpretation that estimates the value ranges of artificial counter variables (see below).

The main part of our low-level analysis is a pipeline analysis coupled with a simulator, see Fig. 1. Currently we can analyse code for NEC V850E [26] and ARM9 [1], but the modular architecture of the tool makes it possible to simply plug in low-level analyses of new processors as the need arises. We have also implemented an instruction cache analysis similar to the one described by Ferdinand et al. [14], but we have not used this analysis in the current experiments. See [9, 10] for details.

Information about possible program flows is communicated to the low-level analysis and WCET calculation through *flow facts*, a certain constraint format [11]. Flow facts are constraints on execution counters that give the number of times certain edges or nodes in the control flow graph are traversed, possibly in the context of an iteration of an enclosing loop. Flow facts are defined relative to *scopes*, basically subgraphs to the control flow graph that constitute loops (or other repeating constructs). Scopes are hierarchically nested as trees. Each scope also has an upper bound on its number of iterations.

The tool can currently use two calculation methods to produce the final WCET bound: an IPET-related method that uses ILP, and a faster but less general path-based method [11].

4 The OSE Operating System

The OSE operating system[1] is a real-time operating system that is used in embedded applications, like in mobile phones and aircrafts. It is one of the major operating systems in the world for these kinds of applications. OSE supports a process model with priorities, where processes can be organized in blocks that have their own shared memory areas. Since it handles concurrent processes with shared resources, there are plenty

[1] `www.ose.com`

of situations where the operating system must be able to complete some task without interruption. Thus, it contains many DI regions. Since it is intended for real-time applications it is important that the execution time of the DI regions is kept short, which motivates the study reported on here.

OSE is available for a number of target processors, mostly towards the high-end spectrum of embedded processors. The delta kernel of OSE, which has been used in this study, is available for ARM, StrongARM, PowerPC, Motorola 68k, and MIPS R3000.

5 The ARM9 Processor

We selected the ARM9 Processor as target architecture, since it is an important architecture to support for OSE, and widely used overall. We implemented a simulator for it during the course of the project.

ARM9 is a 32-bit RISC processor with a five-stage pipeline. It can switch into *THUMB mode*, where it executes a 16-bit instruction set, which is useful in applications where code size is critical. All instructions can be executed conditionally, which can be used to reduce the number of branches in the code.

ARM has three kinds of jump instructions: Branch, Branch-with-link, and Branch-and-exchange. Branch instructions are used when the there is no need to save a return address. Branch-with-link instructions are used for subroutine calls, when the return address must be saved. These instructions take an address as argument, which is saved in a special link register. Branch-and-exchange instructions, finally, are typically used when returning from a subroutine. They take a register as argument, which holds the jump address. This register can be the link register but may also be some other register.

Interrupts are enabled (EI) and disabled (DI) by setting some bits in a status register. This is done with a Move-register-to-status-register instruction. Thus, an EI or DI will be executed depending on the contents in the source register.

6 Experimental Setup

The task at hand was the following: extract the DI regions from the binaries for the kernel, find the control flow graph for these regions, construct *scope graphs* that contain the scope definitions and flow facts for the regions, and finally use the WCET tool to calculate upper bounds for the WCET of each DI region. In order to do this, we had to use a mix of semiautomated and manual routines. We believe this is a fairly typical situation in practice, when tools are to be applied, and thus it could be of some interest to describe this work.

We had access to the object code binaries in ELF (Executable Linkable File) format, and the source code. The ELF files were generated by tools from ARM, and they contain symbol tables which were helpful to relate binary code to its source code. (Alas, this information is non-standard, and thus our tools are dependent on the object code being generated by the ARM tools.) The source code is written both in C and assembler.

We made prototype tools to extract the DI regions, and to build the control flow graph for the extracted regions. The tool to find the regions is an `awk` text processing

script, which filters out the functions in the source code where instructions affecting the interrupt state are detected, and then finds the actual regions by syntactical means. The corresponding binary code is then extracted from the ELF format files. See Fig. 1.

A DI region is defined as the code between a DI and the next EI executed. This definition has some problems, since these regions in principle can be different between different executions. For instance, DI regions may have multiple entries and exits. This can be dealt with by considering all possible paths through the region. The DI regions we found were mostly single entry/single exit, and the few others had a simple control structure, so in practice this was not a problem. (As a side note, we found a few cases where several DI's are executed before any EI: the DI region then starts with the first DI.) A more serious problem was that the instruction used to enable/disable interrupts uses the current value in a register, and thus it may be data-dependent whether an instance of the instruction in the code disables or enables the interrupt! We could identify three categories, depending on the contents of the source register:

- the source register is set or cleared just before the status register is changed,
- the contents of the source register is restored from the stack, and
- the contents of the source register originates from somewhere else.

For the first category, it is easy to decide whether a DI or EI is executed. The second category is typically used to end a DI region, where the current status of the register is saved on the stack before executing the initial DI: the effect is that the interrupt state before entering the region is restored upon exit. A typical situation is when a DI region appears within a subroutine, which should not change the interrupt state of the caller. The third category may appear when, for instance, the value used to set/clear the interrupt enable bits originates from an argument to a function.

Our simple `awk` tool only detects DI regions delimited by DI's and EI's from the first category. Thus, some regions of the OSE delta kernel were not analyzed in the experiment. A more sophisticated tool, that uses a data flow analysis to find the source of the register contents used to set the interrupt bits, could probably find more regions.

A second tool constructs control flow graphs from the extracted binaries for the detected DI regions. This reverse engineering is a necessary step in many analyses of object code, not just WCET analysis of DI regions, and therefore this tool could be used in many other contexts. The tool uses a straightforward algorithm, tracing the jumps in the code, to find the basic blocks and the edges between them. Calls to subroutines and functions are also identified. Here, we had to assume some calling conventions how to use the instruction set, when calling and returning from a subroutine. In particular, we assumed that every Branch-and-exchange operation implemented a return from subroutine, even in cases where the target jump address came from another register than the link register. We also had to handle the fact that some regions of the kernel code execute in THUMB mode. These regions are detected by finding the instructions that switch to and from THUMB mode, much like the detection of DI regions. This was not hard to implement.

Conditionally executed instructions posed a potential problem. In the worst case, one would have to create a separate, conditionally executed basic block for each such instruction, which would lead to a very expensive WCET calculation. Our solution was to simply assume that all instructions (except, of course, jumps) are unconditionally

executed. This may lead to larger WCET estimates, but can never cause an underestimation on the ARM9. (On some other architectures than ARM9 this may actually happen, though, see [9] for how to handle conditional code in general.) An alternative would be to perform a more sophisticated analysis of the conditions, to find sequences of instructions with conditions always evaluating to the same result. These could then be put in the same, conditionally executed basic block.

Finally, the scope graphs were computed. The WCET tool has a module that constructs scope graphs from control flow graphs. This tool essentially identifies loops hierarchically and creates the corresponding scopes. The flow facts for the scopes had to be provided by hand, in the absence of an automatic flow analysis. Of particular importance was to provide upper limits for loop iterations. We believed these could be possible to decide from the source code, but in practice this was hard. A reason for this is that the DI regions often cross function and subroutine boundaries, so the source code for a region can be widely scattered. This made it hard for us to trace back the different parts of DI regions to their origins in the source codes. Another reason for our difficulties was that entities that give upper bounds for the number of loop iterations, like sizes of data structures, were not always explicitly declared in the source code.

The most time-consuming part of this work was to implement the tool that creates control flow graphs from the binary code. However, it also took some time to figure out how to detect DI/EI-instructions.

7 Results

We identified 612 DI regions in the delta OSE kernel. This is approximately half of the total number of interrupt regions, but as explained in Section 6 we collected only those that could be surely identified by our simple tool. Of these, 554 regions contained at most three basic blocks. This confirms the common belief that this kind of region typically is small and has a simple control structure. The belief is further validated by the size distribution of the regions, which is shown in Fig. 2. Loops were found in about 5% of the regions, but rarely more than one loop per region. Only two of the regions contained nested loops, in both cases a single, doubly nested loop. There were no function pointers.

We selected ten DI regions for a closer investigation. We wanted to test regions that were potentially challenging for WCET analysis, with a relatively complex control structure, so several of the regions contain loops. The typical region is shorter, contains no loops, and has few if any branches. The control flow graphs for three of the selected regions are shown in Fig. 3. (All ten are given in [2].)

The estimated WCET's for the selected regions, together with some statistics about their control structure, are shown in Table 1. The unknown upper loop iteration bounds were all set to 100. This is probably a gross overestimation. The estimates were calculated for an ARM9 processor without cache. We used a path-based calculation method. Due to the limited time of this study, we could not perform any real measurements on the execution time for the interrupt regions. Note that DI regions can stretch over function calls: each function call gives a new scope, and therefore the number of loops and the number of scopes may differ.

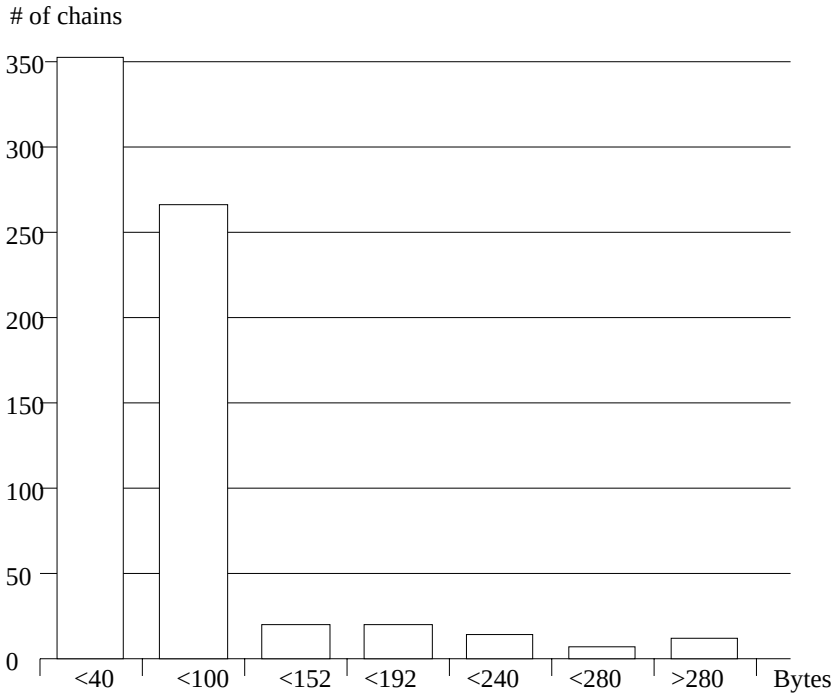

Fig. 2. Size distribution of DI regions.

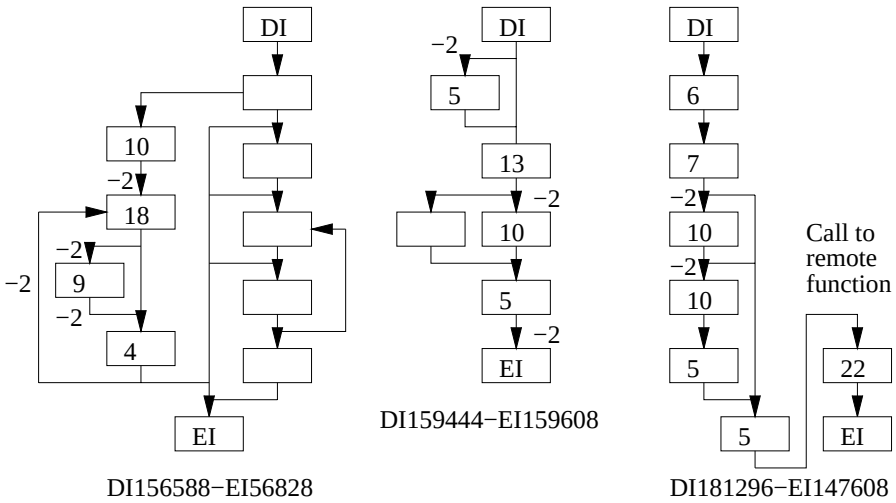

Fig. 3. Control flow graphs for three selected DI regions. The basic blocks in the longest path are annotated with their cycle counts, and the edges with the number of cycles saved from instruction pipeline overlap.

Chain	Size (bytes)	Basic blocks	Loops	Nested	Scopes	WCET (cycles)
DI156588-EI156828	244	11	2	–	4	2564
DI159444-EI159608	168	7	–	–	2	45
DI181296-EI147608	200	9	–	–	2	85
DI182248-EI147608	308	16	–	–	2	89
DI183924-EI183796	160	11	1	–	3	2351
DI183924-EI184392	160	11	1	–	3	2363
DI185276-EI185524	248	11	–	–	2	86
DI194280-EI194376	100	6	–	–	2	31
DI230288-EI230508	240	12	3	1	5	26729
DI261180-EI261432	212	9	3	1	5	26228

Table 1. WCET estimates for ten selected DI regions. For regions with loops, an upper iteration bound of 100 has been assumed except for the inner loops in the two last regions which both were found to have iteration bound 32.

8 Conclusions and Further Research

The most interesting result of this study is not the calculated WCET estimates. They are probably large overestimates of the real WCET on the assumed target system, since the assumed loop bounds most likely are way too large. Also, the ARM9 simulator has not yet been verified against the real hardware (see [9] for how to do this). Furthermore, our target system is an ARM9 system with fast memory and no cache, while the OSE operating system often is used on systems with cache. (Note, however, that potentially long execution times on systems without cache and with cache should correlate positively, so the analysis results might still be useful for detecting potentially long-executing DI regions that are candidates for optimization.) More interesting is the information gained about the typical structure of DI regions: those inspected here turned out to be simple in structure, with few, mostly well-structured loops, very few nested loops, and no function pointers. The last fact is very important since code with function pointers needs a sophisticated analysis to find the possible program flows, with a risk of loss in precision. We therefore do believe that this kind of code is amenable to WCET analysis of the kind we currently are developing.

The case study was done by one person during five months. Due to the limited time, our tool set to find DI regions and construct their control flow graphs has some shortcomings, but most of them seem straightforward to overcome. There is a need for a data flow analysis on low-level code, in order to statically determine the values of registers. Similarly, an analysis to decide whether the conditions for subsequent conditionally executed instructions always evaluate to the same value would also be useful, to find a more detailed control flow graph (and thus a tighter WCET estimate) where the number of basic blocks is still reasonable. There is also a need to make relevant high-level information available on object-code level. Standards for attaching metadata to object code are needed, or tools will become vendor-dependent even when they work on binary code level. Finally, one must map the WCET info back to the source code. This can probably be done using debug info from the compiler used.

It is also clear that the usefulness of analyses like WCET analysis grows fast with the level of automation. In our experiment, even simple means of automation made a huge difference in the amount of engineering work necessary. Besides the analyses already mentioned, we will intensify our efforts to develop an automated flow analysis.

In discussions with Enea OSE Systems, we have learned that absolute WCET bounds seldom are very interesting. The operating systems are often run in certain modes, and the absolute WCET's are often obtained only for very particular modes, which rarely appear in practice (combinations of high levels of error detection, lots of processes, etc.). Rather, one would like to know the WCET conditionally, given that the system runs in a certain mode. Modes, or sets of modes, can often be encoded as value-range constraints on program variables (settings of flags, bounds on number of processes, etc.). Program flow constraints can also be expressed as value-range constraints, but on execution count variables. Thus, it seems of great importance to develop means to communicate such information to the analysis in order to constrain the possible program flows for the given mode. Existing means for bounding the number of loop iterations in WCET calculation do not seem sufficient for this [14, 17, 19, 29, 30]. A more general language to express constraints on variable values would be useful and could also be potentially interesting to use for other program analyses.

9 Acknowledgments

This work was performed within the Advanced Software Technology competence center (ASTEC, `www.docs.uu.se/astec`), supported by the Swedish Agency for Innovation Systems (VINNOVA, `www.vinnova.se`). Direct support was also provided by Enea OSE Systems AB (`www.ose.com`).

References

1. ARM Ltd. *ARM 9TDMI Technical Reference Manual*, 3^{rd} edition, Mar. 2000. Document no. DDI 0180A.
2. M. Carlsson. Worst-Case Execution Time Analysis, Case Study on Interrupt Latency, for the OSE Real-Time Operating System. Master's thesis, KTH/IMIT, Mar. 2002. `ftp://ftp.it.kth.se/Reports/DEGREE-PROJECT-REPORTS/-020326-Martin-Carlsson.pdf`.
3. R. Chapman. Program timing analysis. Dependable Computing System Centre, University of York, England, May 1994.
4. R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94)*, 1994.
5. A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, 18(2/3):249–274, May 2000.
6. A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01)*, pages 191–198, Delft, June 2001. IEEE Computer Society Press.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

8. OSE 4.3 Documentation, Volume 1 – Kernel, 2000. ENEA OSE Systems.
9. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, Apr. 2002. Acta Universitatis Upsaliensis, Dissertations from the Faculty of Science and Technology 36, `http://publications.uu.se/theses/91-554-5228-0/`.
10. J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, Dec. 1999.
11. J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21^{th} IEEE Real-Time Systems Symposium (RTSS'00)*, Nov. 2000.
12. J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 2001. Accepted for publication.
13. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer-Verlag, Aug. 1997.
14. C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.
15. J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Dept. of Information Technology, Uppsala University, May 2000.
16. C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.
17. N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, Sept. 2000.
18. S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proc. 2^{nd} IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 230–240, 1996.
19. Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd ACM IEEE Design Automation Conference (DAC'95)*, pages 456–461, 1995.
20. Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modelling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17^{th} IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263. IEEE Computer Society Press, Dec. 1996.
21. S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.
22. S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS'98)*, Dec. 1998.
23. T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, June 1998.
24. T. Mitra and A. Roychoudhury. Effects of Branch Prediction on Worst Case Execution Time of Programs. Technical Report 11-01, National University of Singapore (NUS), Nov. 2001.
25. F. Mueller. Timing predictions for multi-level caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, Jun 1997.
26. NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3^{rd} edition, Jan. 1999. Document no. U12197EJ3V0UM00.

27. G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.
28. C. Park and A. Shaw. Experiments with a program timing tool based on a source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.
29. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Journal of Real-Time Systems*, 5(1):31–62, Mar. 1993.
30. P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, Apr. 1995.
31. J. Runeson. Code compression through procedural abstraction before register allocation. Master's thesis, Department of Information Technology, Uppsala University, Mar. 2000.
32. J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.
33. F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.
34. F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4^{th} International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001)*. ACM Press, Nov. 2001.
35. H. Theiling. "generating decision trees for decoding binaries". In *"Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)"*, June 2001.
36. R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. 3^{rd} IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, June 1997.