Software Structure and WCET Predictability*

Gernot Gebhard[1], Christoph Cullmann[1], and Reinhold Heckmann[1]

1 AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
info@absint.com

Abstract

Being able to compute worst-case execution time bounds for tasks of an embedded software system with hard real-time constraints is crucial to ensure the correct (timing) behavior of the overall system. Any means to increase the (static) time predictability of the embedded software are of high interest – especially due to the ever-growing complexity of such software systems. In this paper we study existing coding proposals and guidelines, such as MISRA-C, and investigate whether they simplify static timing analysis. Furthermore, we investigate how additional knowledge, such as design-level information, can further aid in this process.

1998 ACM Subject Classification B.2.2 [*Performance Analysis and Design Aids*]: Worst-case analysis

Keywords and phrases WCET Predictability, Embedded Software Structure, Coding Guidelines

Digital Object Identifier 10.4230/OASIcs.PPES.2011.1

1 Introduction

Embedded hard real-time systems need reliable guarantees for the satisfaction of their timing constraints. Experience with the use of static timing analysis methods and the tools based on them in the automotive and the avionics industries is positive. However, both, the precision of the results and the efficiency of the analysis methods are highly dependent on the predictability of the execution platform [3] and of the software run on this platform.

In this paper, we concentrate on the effect of the software on the time predictability of the embedded system. More precisely, we study existing software development guidelines that are currently in production use and identify coding rules that might ease a static timing analysis of the developed software. Such coding guidelines are intended to lead the developer to producing – among others – reliable, maintainable, testable/analyzable, and reusable software. Code complexity is also a key aspect due to maintainability and testability issues. However, the coding rules are not explicitly intended to improve the software predictability with respect to static timing analysis.

Based on our experience of analyzing automotive and avionics software, we provide additional means to increase software time predictability. Certain information about the program behavior cannot be determined statically just from the binary itself (or from the source code, if available). Hence, additional (design-level) knowledge about the system behavior would allow for a more precise (static) timing analysis. For instance, different operating modes of a flight control unit, such as *plane is on ground* and *plane is in air*, might lead to mutual exclusive execution paths in the software system. By using this knowledge, a

* The research reported herein has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement No 216008 (PREDATOR).

static timing analyzer is able to produce much tighter worst-case execution time bounds for each mode of operation separately.

Section 2 discusses related work. Section 3 briefly introduces static timing analysis and discusses challenges static timing analysis has to face. Section 4 investigates existing coding guidelines for their prospects to aid software predictability and discusses further means to increase the predictability of embedded software systems. Finally, Section 5 concludes this paper.

2 Related Work

The impact of the source code structure on time predictability has been subject to several research papers and projects respectively.

For instance, Puschner and Kirner propose a *WCET-oriented programming* approach [11] that aims at producing few or no input-data dependent code. Basically, the idea is to transform the software into a single-path program. To realize input-data dependent behavior of the code – this cannot be avoided for any piece of complex software – predicated operations shall be used[1]. A major drawback of the proposed code transformation is that in every possible execution context of a function or loop, the processor would have to always fetch the corresponding instructions, even if they would not be executed. Hence, the single-path paradigm actually impairs the worst-case behavior.

Thiele and Wilhelm investigate threats to time predictability and propose design principles that support time predictability [14]. Among others the authors discuss the impact of software design on system predictability. For example, the use of dynamic data structures should be avoided, as these are hard to analyze statically.

Wenzel et al. [15] discuss the possible impact of existing software development guidelines (DO-178B, MISRA-C, and ARINC 653) on the WCET analyzability of the software. Furthermore, the authors provide challenging code patterns, some of which, however, do not appear to cause problems for binary-level, static WCET analysis. For instance, calls to library functions do not necessarily impair the software's time predictability. The implementation and thus the binary code of the called function determines the time predictability, and not the fact of the function being part of a library. Nonetheless, the binary code of the library functions are required to be available to ensure a precise static worst-case execution time analysis if complex hardware architectures are being used. For ARINC 653 implementations that are truly modular this might not always be the case.

The purpose of the project COLA (Cache Optimizations for LEON Analyses)[2] was to investigate how software can achieve maximum performance, whilst remaining analyzable, testable, and predictable. COLA is a follow-on project to the studies PEAL and PEAL2 (Prototype Execution-time Analyzer for LEON), which identified code layout and program execution patterns that result in cache risks, so called *cache killers*, and quantified their impact. Among others, the COLA project produced cache-aware coding rules that are specifically tailored to increase the time predictability of the LEON2 instruction cache.

The project MERASA aimed at the development of a predictable and (statically) analyzable multi-core processor for hard real-time embedded systems. Bonenfant et al. [1] propose coding guidelines to improve the analyzability of software executed on the MERASA platform.

[1] Yet many embedded hardware architectures, as e.g. PowerPC, do not support predicated operations.

[2] Funded by the European Space Agency (ESA) under the basic Technology Research Programme (TRP), ESA/ESTEC Contract AO/1-5877/08/NL/JK.

Both static analysis and measurement-based approaches are considered. In principle, these coding guidelines correspond to the MISRA-C guidelines discussed in Section 4.2.

3 Static Timing Analysis

Exact worst-case execution times (WCETs) are impossible or very hard to determine, even for the restricted class of real-time programs with their usual coding rules. Therefore, the available WCET analyzers only produce WCET guarantees, which are safe and precise upper bounds on the execution times of tasks. The combined requirements for timing analysis methods are:

- *soundness* – ensuring the reliability of the guarantees,
- *efficiency* – making them feasible in industrial practice, and
- *precision* – increasing the chance to prove the satisfaction of the timing constraints.

Any software system when executed on a modern high-performance processor shows a certain variation in execution time depending on the input data, the initial hardware state, and the interference with the environment. In general, the state space of input data and initial states is too large to exhaustively explore all possible executions in order to determine the exact worst-case and best-case execution times. Instead, bounds for the execution times of basic blocks are determined, from which bounds for the whole system's execution time are derived.

Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information, and thus are – in part – responsible for the gap between WCET guarantees and observed upper bounds and between BCET guarantees and observed lower bounds. How much is lost depends on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software.

Despite the potential loss of precision caused by abstraction, static timing analysis methods are well established in the industrial process, as proven by the positive feedback from the automotive and the avionics industries. However, to be successful, static timing analysis has to face several challenges, being discussed in the subsequent Section 3.2.

3.1 Tools for Static Worst-Case Execution Time Analysis

Figure 1 shows the general structure of WCET analyzers like aiT, see `http://www.absint.com/aiT` – this is the static WCET tool we are most experienced with. The input binary executable has to undergo several analysis phases, before a worst-case execution time bound can be given for a specific task[3]. First, the binary is decoded (reconstruction of the control-flow). Next, loop and value analysis try to determine loop bounds and (abstract) contents of registers and memory cells. The (cache and) pipeline analysis computes lower and upper basic block execution time bounds. Finally, the path analysis computes the worst-case execution path through the analyzed program (see [3] for a more detailed explanation).

3.2 Challenges

A static WCET analysis has to cope with several challenges to be successful. Basically, we discern two different classes of challenges. Challenges that need to be met to make the

[3] A task (usually) corresponds to a specific entry point of the analyzed binary executable.

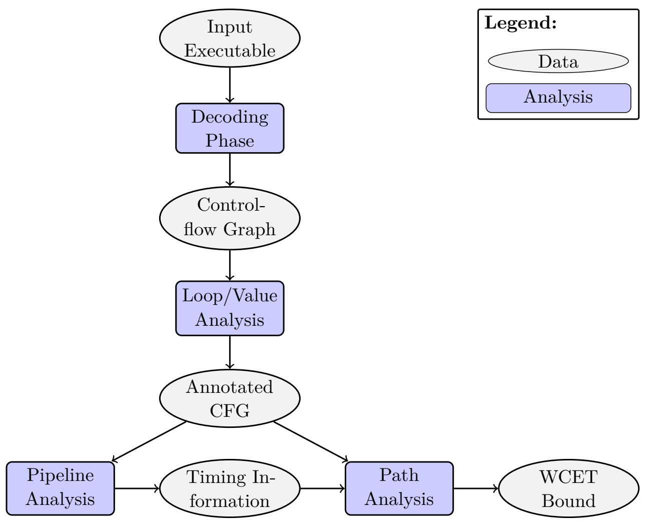

Figure 1 Phases of WCET computation.

WCET bound computation feasible at all are *tier-one* challenges. *Tier-two* challenges are concerned with keeping the WCET bounds as tight as possible, e.g., to enable a feasible schedule of the overall system.

The used coding style is tightly coupled to the encountered tier-one challenges. Section 4 investigates whether coding guidelines that are in production use (indirectly) address such challenges and whether they ease their handling. Section 4.3 provides means how to cope with tier-two challenges.

In the following, we discuss tier-one WCET analysis challenges.

Function Pointers. Often simple language constructs do not suffice to implement a certain program behavior. For instance, user-defined event handlers are usually implemented via function pointers to exchange data between communication library (e.g., for CAN devices) and the application. Resolving function pointers automatically is not easily done and sometimes not feasible at all. Nevertheless, function pointers need to be resolved to enable the reconstruction of a valid control-flow graph and the computation of a WCET bound.

Loops and Recursions. Loops (and also recursions) are a standard concept in software development. The main challenge is to automatically bound the maximum possible number of loop iterations, which is mandatory to compute a WCET bound at all. Whereas often-used counter loops can be easily bounded, it is generally infeasible to bound input-data dependent loops without additional knowledge. Similarly, such knowledge is required for recursions.

Irreducible Loops. Usually, loops have a single entry point and thus a single loop header. However, more complicated loops are occasionally encountered. By using language constructs like the `goto` statement from C or by means of hand-written assembly code, it is possible to construct loops featuring multiple entry points. So far, there exists no feasible approach to automatically bound this kind of loops [8]. Hence, additional knowledge about the control-flow behavior of such loops is always required.

4 Software Predictability

In this section we discuss existing coding standards and investigate rules from the 2004 MISRA-C standard that are beneficial for software predictability. Thereafter, we describe how design-level information can further aid static timing analysis.

4.1 Coding Guidelines

Several coding guidelines have emerged to guide software programmers to develop code that conforms to safety-critical software principles. The main goal is to produce code that does not contain errors leading to critical failure and thus causing harm to individuals or to equipment. Furthermore, software development rules aim at improved reliability, portability, and maintainability.

In 1998, the Motor Industry Software Reliability Association (MISRA) published MISRA-C [9]. The guidelines were intended for embedded automotive systems implemented in the C programming language. An updated version of the MISRA-C coding guidelines has been released in 2004 [10]. This standard is now widely accepted in other safety-critical domains, such as avionics or defense systems. On the basis of the 2004 MISRA-C standard, the Lockheed Martin Corporation has published coding guidelines that are obligatory for Air Vehicle C++ development in 2005 [2]. Albeit certain rules tackle code complexity, there are no rules that explicitly aim at developing better time predictable software.

4.2 MISRA-C

Wenzel et al. [15] reckon that among the standards DO-178B, MISRA-C, and ARINC 653, only MISRA-C includes coding rules that can effect software predictability. In the following, we thus take a closer look at the 2004 MISRA-C guidelines. The list partially corresponds to the one found in [15] (focusing on 1998 MISRA-C), but refers to the potential impact on the time predictability using binary-level static WCET analysis (e.g., with the aiT tool).

Rule 13.4 (required): ***The controlling expression of a for statement shall not contain any objects of floating type.*** State-of-the-art abstract interpretation based loop analyzers work well with integer arithmetic, but do not cope with floating point values [5, 4]. Thus, by forbidding floating point based loop conditions, a loop analysis is enabled to automatically detect loop bounds.

Rule 13.6 (required): ***Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.*** This rule promotes the use of (simple) counter-based loops and prohibits the implementation of a complex update logic of the loop counter. This allows for a less complicated loop bound detection.

Rule 14.1 (required): ***There shall be no unreachable code.*** Tools like aiT can detect that some part of the code is not reachable. However, static timing analysis computes an over-approximation of the possible control-flow. By this, the analysis might assume some execution paths that are not feasible in the actual execution of the software. Hence, the removal of unreachable parts from the code base leads to less sources of such imprecision.

Rule 14.4 (required): ***The goto statement shall not be used.*** The usage of the `goto` statement does not necessarily cause problems for binary-level timing analysis. These statements are compiled into unconditional branch instructions, which are no challenge to such analyses by themselves. However, the usage of the `goto` statement might possibly introduce irreducible loops into the program binary. There is no known approach available to automatically determine loop bounds for this kind of loops. Consequently, manual annotations are always required. Even worse, certain precision-enhancing analysis techniques, such as virtual loop unrolling [13], are not applicable.

Rule 14.5 (required): ***The continue statement shall not be used.*** Wenzel et al. [15] state that not adhering to this rule could lead to unstructured loops (see rule 14.4). However, `continue` statements only introduce additional back edges to the loop header and therefore cannot lead to irreducible loops. Any loop containing `continue` statements can be transformed into a semantically equivalent loop by means of `if-then-else` constructs. Hence, the only purpose of this rule is to enforce a certain coding style.

Rule 16.1 (required): ***Functions shall not be defined with a variable number of arguments.*** Functions with variable argument lists inherently lead to data dependent loops iterating over the argument list. Such loops are hard to bound automatically.

Rule 16.2 (required): ***Functions shall not call themselves, either directly or indirectly.*** Similarly to using `goto` statements, the use of recursive function calls might lead to irreducible loops in the call graph. Thus, a similar impact on software predictability would apply as discussed above for `goto` statements (see rule 14.4).

Rule 20.4 (required): ***Dynamic heap memory allocation shall not be used.*** Dynamic memory allocation leads to statically unknown memory addresses. This will lead to an over-estimation in the presence of caches or multiple memory areas with different timings. Recent work tries to address this problem by means of cache-aware memory allocation [6].

Rule 20.7 (required): ***The setjmp macro and the longjmp function shall not be used.*** In accordance to the discussion of rule 14.4 and of rule 16.2, the usage of the `setjmp` and the `longjmp` macro would allow the construction of irreducible loops. Hence, similar time predictability problems would arise.

4.3 Design-Level Information

Coping with all tier-one challenges of WCET analysis (see Section 3.2) is usually not sufficient in industrial practice. Additional information that is available from the design-level phase is often required to allow a computation of significantly tighter worst-case execution time bounds. Here, we address the most relevant tier-two challenges.

Operating Modes. Many embedded control software systems have different operating modes. For example, a flight control system differentiates between flight and ground mode. Any such operating mode features different functional and therefore different timing behavior. Unfortunately, the modes of behavior are not well represented in the control software code. Although there is ongoing work to semi-automatically derive operating modes from the source code [7], we still propose to methodically document their behavioral impact.

Such documentation could include loop bounds or other kinds of annotations specific to the corresponding operating mode. At best developers should instantly document the relevant source code parts to avoid a later hassle of reconstructing this particular knowledge.

Complex Algorithms. Complex algorithms or state machines are often modeled with tools like MATLAB or SCADE. By means of code generators these models are then transferred into, e.g., C code. During this process, high-level information about the algorithm or the state machine update logic respectively are lost (e.g., complex loop bounds, path exclusions).

Wilhelm et al. [16] propose systematic methods to make model information available to the WCET analyzer. The authors have successfully applied their approach and showed that tighter WCET bounds are achievable in this fashion.

Data-Dependent Algorithms. Computing tight worst-case execution time bounds is a challenging task for strongly data-dependent algorithms. This is mainly caused by two reasons. *On the one hand*, data-dependent loops are hardly bounded statically. However, for computing precise WCET bounds, it generally does not suffice to assume the maximal possible number of loop iterations for each execution context. *On the other hand*, a static analysis is often unable to exclude certain execution paths through the algorithm without further knowledge about the execution environment. The following example demonstrates this problem.

Message-based communication is usually implemented by means of fixed-size read and write buffers that are reserved for each scheduling cycle separately. During an interrupt handler the message data is either copied from or to memory – depending on the current scheduling cycle. Here, read and write operations can never occur in the same execution context of the message handler. Without further information both operations cannot be excluded by a static WCET analysis. Additionally, the analysis has no a-priori information about the amount of data being transferred. However, the allocation of the data buffers and the amount of data to transmit is statically known during the software design phase. Using this information would allow for a much more precise static timing analysis of such algorithms.

Imprecise Memory Accesses. Unknown or imprecise memory access addresses are one of the main challenges of static timing analysis for two reasons. *First*, they impair the precision of the value analysis. Any unknown read access introduces unknown values into the value analysis and therefore increases the possibly feasible control-flow paths and negatively influences the loop bound analysis. In addition, any write access to an unknown memory location destroys all known information about memory during the value analysis phase. *Second*, the pipeline analysis has to assume that any memory module might be the target of an unknown memory access – the slowest memory module will thus contribute the most to the overall WCET bound. For architectures featuring data caches, an imprecise memory access invalidates large parts of the abstract cache (or even the whole cache) and leads to an over-approximation of the possible cache misses on the WCET path. Such unknown memory accesses can result from the extensive use of pointers inside data structures with multiple levels of indirections.

A remedy to this could be to document the memory areas that might be accessed for each function separately, especially if slow memory modules could be accessed. For example, memory-mapped I/O regions that are used for CAN or FLEXRAY controllers usually are only accessed in the corresponding device driver routines. Thus, the analysis would only

Iteration Counts	Frequency of Occurrence	Observed for
0	1 552	
1	99 881 801	
2	116 421	
3	114	
4 .. 9	13	
10 .. 19	19	
20 .. 39	24	
40 .. 59	22	
60 .. 79	13	
80 .. 99	11	
100 .. 135	7	
156	1	`lDivMod (0x ffd9 3580, 0x 107 d228)`
186	1	`lDivMod (0x fff2 c009, 0x 118 dcc4)`
204	1	`lDivMod (0x ffe8 70e3, 0x 141 4167)`

Table 1 Observed iteration counts for `lDivMod`.

need to assume for those specific routines that imprecise or unknown memory accesses target these (slow) memory regions. For all other routines, the analysis would be allowed to assume that different, potentially faster memory modules are being accessed.

Error Handling. In embedded software systems, error handling and recovery is a very complex procedure. In the event of an error, great care needs to be taken to ensure safety for individuals and machinery respectively.

A precise (static) timing analysis of error handling routines requires a lot more than the maximum number of possible errors that can occur or have to be handled at once. First of all however, it needs to be decided whether the error case is relevant for the worst-case behavior or not. If not, all error-case related execution paths through the software may be ignored during WCET analysis, which will obviously lead to much lower WCET bounds being computed. This however requires precise knowledge about which parts of the software are concerned with handling errors.

Otherwise, static timing analysis has to cope with error handling. The assumption that all errors might occur at once naturally leads to safe timing guarantees. However, in reality this is a rather uncommon or simply infeasible behavior of the embedded system. Here, computing tight WCET bounds requires precise knowledge about all potential error scenarios. An early documentation of the system's error handling behavior is thus expected to allow for a quicker and more precise analysis of the overall system.

Software Arithmetic. Under certain circumstances, an embedded software system makes use of software arithmetic. This is the case if the underlying hardware platform does not support the required arithmetic capabilities. For instance, the Freescale MPC5554 processor only supports single precision floating point computations [12]. If higher-precision FPU operations are required, (low-level) software algorithms emulating the required arithmetic precision come into play. Such algorithms are usually designed to provide good average-case performance, but are not implemented with good WCET predictability in mind. This often causes a static timing analysis to assume the worst-case path through such routines for most execution contexts.

An extreme example for a function with good average-case performance and bad WCET predictability is the library function `lDivMod` of the CodeWarrior V4.6 compiler for Freescale HCS12X. The purpose of this routine is to compute quotient and remainder of two 32 bit unsigned integers. The algorithm performs an iteration computing successive approximations to the final result. To get an impression on the number of loop iterations, we performed an experiment in which `lDivMod` was applied to 10^8 random inputs. Table 1 shows which iteration counts were observed in this experiment. The number of iterations is 1 in more than 99.8% and 0, 1, or 2 in more than 99.999% of the sample inputs. On the other hand, iteration counts of more than 150 could be observed for a few specific inputs. There seems to be no simple way to derive the number of iterations from given inputs (other than running the algorithm). The highest possible iteration count could not yet be determined by mathematical analysis. Even if it were known that 204 is the maximum, a worst-case execution time analysis had to assume that such a high iteration number occurs when the input values cannot be determined statically, leading to a big over-estimation of the actual WCET.

To tighten the computed WCET bounds, further information would be required to avoid the cases with high numbers of loop iterations in many or all execution contexts. Making sure that the used software arithmetic library features good WCET analyzability also helps to tighten the computed WCET bounds. Another – more radical – approach would be to employ a different hardware architecture that supports the required arithmetic precision.

5 Conclusion

Our experience with static timing analysis of embedded software systems shows that the analysis complexity varies greatly. As discussed above, the software structure strongly influences the analyzability of the overall system. Existing coding guidelines, such as the MISRA-C standard, partially address tier-one challenges encountered during WCET analysis. However, solely adhering to these guidelines does not suffice to achieve worst-case execution time bounds with the best precision possible. We usually suggest to document the software system behavior as early as possible – desirably during the software design phase – to tackle the tier-two WCET analysis challenges. Otherwise, achieving precise analysis results during the software development testing and validation phase might become a costly and time consuming process.

References

1 Armelle Bonenfant, Ian Broster, Clément Ballabriga, Guillem Bernat, Hugues Cassá, Michael Houston, Nicholas Merriam, Marianne de Michiel, Christine Rochange, and Pascal Sainrat. Coding guidelines for WCET analysis using measurement-based and static analysis techniques. Technical Report IRIT/RR–2010-8–FR, IRIT, Université Paul Sabatier, Toulouse, March 2010.

2 Lookheed Martin Corporation. C++ coding standards for the system development and demonstration program, December 2005.

3 Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:26–42, 2010.

4 Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *Workshop on Worst-Case Execution-Time Analysis (WCET)*, volume 6 of *OASICS*, July 2007.

5 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *Workshop on Worst-Case Execution-Time Analysis (WCET)*, volume 6 of *OASICS*, July 2007.

6 Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-aware memory allocation for WCET analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.

7 Philipp Lucas, Oleg Parshin, and Reinhard Wilhelm. Operating mode specific WCET analysis. In Charlotte Seidner, editor, *Proceedings of JRWRTC*, October 2009.

8 Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the International Conference on Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

9 The Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in vehicle based software, 1998.

10 The Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in critical systems, October 2004.

11 Peter Puschner and Raimund Kirner. Avoiding timing problems in real-time software. In *1st IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES 2003)*. IEEE Computer Society, 2003.

12 Freescale Semiconductor. e200z6 PowerPC Core Reference Manual, 2004.

13 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2–3):157–179, 2000.

14 Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157–177, 2004.

15 Ingomar Wenzel, Raimund Kirner, Martin Schlager, Bernhard Rieder, and Bernhard Huber. Impact of dependable software development guidelines on timing analysis. In *Proceedings of the 2005 IEEE Eurocon Conference*, pages 575–578, Belgrad, Serbia and Montenegro, 2005. IEEE Computer Society.

16 Reinhard Wilhelm, Philipp Lucas, Oleg Parshin, Lili Tan, and Björn Wachter. Improving the precision of WCET analysis by input constraints and model-derived flow constraints. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, LNCS. Springer-Verlag, 2010. To appear.