# Static Timing Analysis for Hard Real-Time Systems

Sebastian Altmeyer, Mohamed Abdel Maksoud, Claire Burguiere, Daniel Grund, Jörg Herter, Philipp Lucas, Oleg Parshin, Markus Pister, Jan Reineke, Marc Schlickling, Björn Wachter, Reinhard Wilhelm

Compiler Design Lab, Saarland University

**Abstract.** Hard real-time systems impose strict timing constraints. To prove that these constraints are met, timing analyses aim to derive safe upper bounds on a task's execution time. Especially modern processor features (caches, out-of-order pipelines, etc.) have a strong impact on the variation of a task's execution time and thus, on the precision and the complexity of the timing analysis. Naive or measurement-based approaches usually can not guarantee safe and reliable timing bounds.
This paper provides an overview of the current timing analysis technique and shortly present ongoing research at the Compiler Design Lab at Saarland University. An extended version can be found in [Wil05].

## 1 Static Timing Analysis

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. This article treats timing analysis of tasks with uninterrupted execution. 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 WCETs and upper bounds and between BCETs and lower bounds. How much is lost depends both on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software. The methods used to determine upper bounds and lower bounds are very similar.

In modern microprocessor architectures, caches, pipelines, and all kinds of speculation are key features for improving (average-case) performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution time of individual instructions, and thus the contribution to the program's execution time can

vary widely. The interval of execution times for one instruction is bounded by the execution times of the following two cases:

- The instruction goes "smoothly" through the pipeline; all loads hit the cache, no pipeline hazard happens, i.e., all operands are ready, no resource conflicts with other currently executing instructions exist.
- "Everything goes wrong", i.e., instruction and/or operand fetches miss the cache, resources needed by the instruction are occupied, etc.

We will call any increase in execution time during an instruction's execution a *timing accident* and the number of cycles by which it increases the *timing penalty* of this accident. Timing penalties for an instruction can add up to several hundred processor cycles. Whether the execution of an instruction encounters a timing accident depends on the execution state, e.g., the contents of the cache(s), the occupancy of other resources, and thus on the execution history. It is therefore obvious that the attempt to predict or exclude timing accidents needs information about the execution history.

### 1.1 Timing Analysis Framework

Over the last several years, a more or less standard architecture for timing-analysis tools has emerged [HWH95,TFW00,Erm03]. 1 gives a general view on this architecture. First, one can distinguish three major building blocks:

- Control-flow reconstruction and static analyses for control and data flow.
- Micro-architectural analysis, which computes upper and lower bounds on execution times of basic blocks.
- Global bound analysis, which computes upper and lower bounds for the whole program.

The following list presents the individual phases and describes their objectives and problems. Note that the first four phases are part of the first building block.

1. *Control-flow reconstruction* [The02a] takes a binary executable to be analyzed, reconstructs the program's control flow and transforms the program into a suitable intermediate representation. Problems encountered are dynamically computed control-flow successors, e.g. stemming from switch statements, function pointers, etc..
2. *Value analysis* [CC77,SS07] computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. This information is among others used for a precise data-cache analysis.
3. *Loop bound analysis* [EG97,HSR+00] identifies loops in the program and tries to determine bounds on the number of loop iterations, information indispensable to bound the execution time. Problems are the analysis of arithmetic on loop counters and loop exit conditions, as well as dependencies in nested loops.

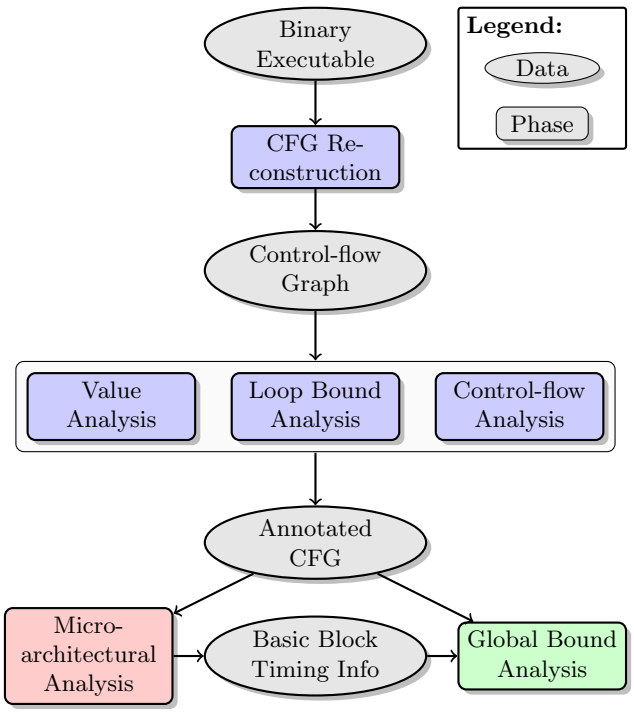


**Fig. 1.** Main components of a timing-analysis framework and their interaction.

4. *Control-flow analysis* [EG97,SM07a] narrows down the set of possible paths through the program by eliminating infeasible paths or to determine correlations between the number of executions of different blocks using the results of value analysis results. These constraints will tighten the obtained timing bounds.
5. *Micro-architectural analysis* [Eng02,The04,FW99] determines bounds on the execution time of basic blocks by performing an abstract interpretation of the program, taking into account the processor's pipeline, caches, and speculation concepts. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc.. Ignoring these average-case-enhancing features would result in imprecise bounds.
6. *Global bound analysis* [LM95,The02b] finally determines bounds on execution time for the whole program. Information about the execution time of basic blocks is combined to compute shortest and longest paths through the program. This phase takes into account information provided by the loop bound- and control-flow analysis.

The commercially available tool `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm`, implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [FW99,FHL+01,TSH+03,HLTW03].

### 1.2 Timing Anomalies

Most powerful microprocessors have so-called *timing anomalies*. Timing anomalies are contra-intuitive influences of the (local) execution time of one instruction on the (global) execution time of the whole program. Several processor features can interact in such a way that a locally faster execution of an instruction can lead to a globally longer execution time of the whole program.

For example, a cache miss contributes the cache-miss penalty to the execution time of a program. It was, however, observed for the MCF 5307 [RSW02], that a cache miss may actually speed up program execution. Since the MCF 5307 has a unified cache and the fetch and execute pipelines are independent, the following can happen: A data access that is a cache hit is served directly from the cache. At the same time, the pipeline fetches another instruction block from main memory, performing branch prediction and replacing two lines of *data* in the cache. These may be reused later on and cause two misses. If the data access was a cache miss, the instruction fetch pipeline may not have fetched those two lines, because the execution pipeline may have resolved a misprediction before those lines were fetched.

## 2 Work In Progress

In this section, we shortly present recent or ongoing work from our group. This research either aims at reducing the shortcomings of the timing analysis or try to broaden its applicability.

### 2.1 Constraints on the Control Flow Graph

There are three influences on the execution time of a task:

1. *Task inputs* and *logical state* determine the dynamical sequence of instructions and memory accesses;
2. the *dynamic hardware state* (cache contents, pipeline states) determines how instructions are executed;
3. *static hardware properties* such as length of pipelines and speed of execution units determine the actual execution time.

Whereas before we have considered how to cope with uncertainty in (2) and how to model (3), we now consider ways to improve precision of WCET calculations by restricting the uncertainty arising from (1). We briefly sketch the general ideas; for a deeper explanation, see [WLP+10].

A WCET analysis tool has to consider the worst-case path through the control-flow graph. There are various ways of conceptually restricting the set of possible paths and of expressing these restrictions.

**Model-based flow constraints** On the one hand, the WCET tool might regard a path as feasible, and thus potentially a worst-case path, which actually is infeasible[1]. Although there is work on detecting infeasible paths in the executable to be analysed (e.g., [SM07b]), the scope of the analysis is only a *single* run through the task. Correlations between control flow choices such as `a>0` and `a<=0` may be detected in this way. However, correlations between variables which arise from the execution of a task in a loop (*inter-run* properties) cannot be determined by the analysis.

For example, if control logic is implemented by automata in a high-level design tool such as Stateflow/Simulink, not all combinations of automata states are reachable. This restriction on control state in turn restricts the possible behaviours of the model, which translates to restrictions on the control flow path through the model's implementation. Analysis of Stateflow automata and their effect on the Simulink model they control thus can lead to control flow restrictions. These restrictions may be expressed to the WCET tool as *flow constraints* or generalisations thereof: Flow constraints introduce additional inequalities relating the execution counts of basic blocks in the ILP which determines the actual worst-case path.

[1] That is, it cannot happen during any execution, though it does not contain any dead code.

**Mode-specific input constraints** Another way of restricting the control flow is by considering *mode-specific* behaviours. Embedded control systems often have radically different *behaviours* depending on their operating mode (e.g., start-up, running, shut-down and error). This concerns for example their WCETs or accesses to shared ressources. Modes also may impose specific *requirements* on the tasks such as mode-specific deadlines. An accurate analysis thus requires the determination of mode-specific WCETs.

Mode-specific WCET analysis strives to infer operating modes from source code, to identify those operating modes with significantly differing WCETs and to compute mode-specific WCETs. To this end, an analysis of the syntactical and semantical properties of a C program heuristically infers operating modes by considering code patterns and mode-signifying differences in behaviour. The determination of mode-specific WCET bounds can then be done by annotations in several ways. Again, flow constraints can be employed to model the *effect* of a mode on the possible control flows. Furthermore, *input constraints* can communicate to the tool the set of input variables giving rise to a particular mode. For example, in one mode a sensor variable may be fixed to values $[0, \infty)$ and in another one (an error mode) to $-1$. Such annotations are used in the earlier value analysis phase of the system. Therefore they effect the complete WCET analysis and lead to more specific results, in contrast to flow annotations which effect only the latest analysis phase.

### 2.2 Semi-automatic derivation of Pipeline Analyses from VHDL Models

Currently, the pipeline models are *hand-crafted* by human experts [The04]. Therefore the model creation as well as the implementation of the corresponding pipeline analysis is a very time-consuming and error-prone process.

As modern processors are synthesized out of formal hardware description languages, like *VHDL*, in which their behavior (including the timing) is exactly specified, the timing model could be semi-automatically derived from it. This would avoid errors introduced by manual implementations due to human involvement and it would speed up the process, too.

VHDL models of real world processors are usually very big and complex. Just generating a pipeline analysis that covers the whole micro-architectural behavior[2] would additionally increase the state space. This would render the resulting timing analysis infeasible in terms of space and time consumption.

We want to derive computational feasible timing models out of formal *VHDL* specifications in a semi-automatic way. In a first step, we reduce the size of the model by pruning out all parts that does not contribute to the timing behavior. For example, we don't need information about each step within a multiplier unit. Instead, it suffices to know how many clock cycles an instruction occupies each stage of the multiplier pipeline.

[2] in this case, the timing model would be equivalent to the full *VHDL* model

The pruned model still contains a lot of detailed information about the processor state. But for practical reasons it is impossible to represent all state information in full detail. If we were to exactly record e.g. the contents of all memory cells or registers, the space required for the analysis would be prohibitive. Luckily, in many cases the exact knowledge about these things is not important as far as timing is concerned: an addition always takes the same amount of time, no matter what the arguments are. In other cases, the timing does depend on such information, but we may choose to lose the exact timing knowledge in order to make the analysis more efficient, or even to make it possible at all. One example for this are multiplications on some architectures, which are faster if one argument has many leading zero bits. By not keeping track of the arguments exactly, we have to assume an entire range of execution times for multiplication. The loss in precision is acceptable in this case, as the difference is usually only a few processor cycles and multiplications are rare.

Therefore, the second part of the timing model derivation is the definition of abstractions on the processor state. Abstractions means that we either left out some details of the processor state or we approximate them. An example for an approximation is the replacement of concrete addresses by address intervals. For memory accesses, we do not need to know the exact address. We only need to know the type of memory that is accessed in order to simulate its timing behavior.

Using the methodology of abstract interpretation, one can trade precision of the analysis against efficiency by choosing different processor abstractions and concretization relations between the concrete processor state and the abstract one.

### 2.3 FIFO Cache Analysis

Precise and efficient analyses have been developed for set-associative caches that employ the least-recently-used ($LRU$) replacement policy [Alt96,FW99]. Generally, research in the field of embedded real-time systems assumes $LRU$ replacement. In practice however, other policies like first-in first-out ($FIFO$) or pseudo-$LRU$ ($PLRU$) are also commonly used, e.g. in the *Intel XScale*, some *ARM9* and *ARM11*, and the *PowerPC 75x* series.

Two kinds of information can be naturally distinguished in cache analysis: must-information that allows for predicting hits, and may-information that allows for predicting misses. Previous work showed that it is inherently more difficult to obtain may-information for than for $LRU$; see [RGBW07]. A first step towards the analysis of those policies was the general concept of relative competitiveness; see [Rei08]. Depending on the particular policy, however, a cache analysis based on relative competitiveness may be anything from very precise to ineffective.

In [GRG09], we describe a generic policy-independent framework for cache analysis. It allows for cooperation of may- and must-analyses through a minimal interface, which improves their precision.

In addition, we present a may- and a must-analysis for *FIFO* caches. The must-analysis borrows basic ideas from *LRU*-analysis [FW99]. To predict cache hits, it infers upper bounds on cache misses to prove containedness of memory blocks. To predict cache misses, the may-analysis infers lower bounds on cache misses to prove eviction. By taking into account the order in which hits and misses happen, we improve the may-analysis, thereby increasing the number of predicted cache misses. Through the cooperation of the two analyses in the generic framework, this also improves the precision of the must-analysis.

### 2.4 Branch Target Buffer

In today's systems, caches, deep pipelines, BTBs, and all sorts of speculation are used to increase average-case performance. These features are challenging for timing analysis since they cause a large variability in the execution times of instructions. If an analysis cannot safely exclude spurious detrimental behavior (cache misses, pipeline stalls, etc.) the obtained WCET bounds may become imprecise and thus useless. It depends on the design of the hardware components how well analyses can exclude such behavior.

BTBs cache addresses of branch targets or instructions at branch targets to reduce the latency when processing branches. Branches occur relatively often and the difference in latency between a BTB hit and a miss is large enough to have a significant influence on the execution time. Thus, a BTB analysis is necessary to obtain precise WCET bounds.

We introduce a modular WCET analysis framework for BTBs that can be adapted to various BTB implementations. It consists of a fixed main module that is the same for all BTBs and two parameter modules each of which answers one of the following questions: For which branches does the BTB contain information? What information is stored in the BTB for a given branch? The modules interact via fixed interfaces such that they can be exchanged independently.

Our second contribution is an instantiation of our framework for the *PowerPC*. The *PowerPC* is used in time-critical automotive and avionics systems and features a branch processing unit (BPU) with branch prediction and a BTB. This case study shows the applicability of our framework for a non-trivial case and demonstrates the effort needed to model a hardware feature. This instantiation improves the WCET bounds by on average 13% on a set of avionic and compiler benchmarks. On a subset of the benchmarks measuring execution time was possible, which yields under-approximations of the WCET but allows to bound the overestimation of our analysis. For this subset, our analysis reduced the average overestimation from 54% to 20%.

Our last contribution is the identification of principles of more predictable hardware designs and their influence on WCET bounds. We identify problems regarding predictability of the example BTB we study. Capitalizing on the modularity of our analysis, we propose alternative hardware designs and evaluate them by additional experiments. In case of the *PowerPC*, employing a more predictable replacement policy (*LRU*) in the BTB would improve the computed

WCET bounds by 2.9% and reduce analysis time considerably. Minor modifications that increase uniformity by eliminating special cases would not only simplify analysis but also improve the WCET bounds obtained with our analysis by up to 20%. Finally, we generalize our findings and give advice to hardware designers.

### 2.5 Heap-allocating Programs

Static worst-case execution time analyses rely on high cache predictability in order to achieve precise bounds on a program's execution time. Such analyses, however, fail to cope with programs using dynamic memory allocation. This is due to the unpredictability of the cache behavior introduced by the dynamic memory allocators.

We investigate two approaches to enable precise worst-case execution time analysis for programs that use dynamic memory allocation.

The first approach automatically transforms the dynamic memory allocation into a static allocation with comparable memory consumption [HR09]. Hence, we try to preserve the main advantage of dynamic memory allocation, namely the reduction of memory consumption achieved by reusing deallocated memory blocks for subsequent allocation requests. However, ending up with a static allocation schemes allows for using existing techniques for WCET analyses.

The second approach replaces the unpredictable dynamic memory allocator by a predictable dynamic allocation [HRW08].

Both approaches rely on precise information about the dynamically allocated heap objects and data structures arising during program executions. Obtaining this information requires an adapted shape analysis together with appropriate abstraction techniques.

### 2.6 BDDs to improve the Pipeline Analysis

Static timing analysis only becomes computationally feasible in practice by using abstraction, which is applied to both the modeling of processor and program behavior. However, abstraction loses information which leads to uncertainty, e.g. it may not be possible to statically determine the exact address of a memory access. Furthermore, program inputs are not precisely known in advance. At the level of the hardware model, this lack of information is accounted for by non-deterministic choices. To be safe, the analysis has to exhaustively explore all possibilities. This can lead to state explosion making an explicit enumeration of states infeasible due to memory and computation time constraints.

We address the state explosion problem in static WCET analysis by storing and manipulating hardware states in a more efficient data structure based on Ordered Binary Decision Diagrams (BDDs). Our work is inspired by BDD-based symbolic model checking. Symbolic model checking has been successfully applied to components of processors. Its success sparked a general interest in BDDs and other symbolic representations. Today, BDDs are also used extensively to analyze software, e.g. in software model checking and points-to analysis.

We enhance the existing framework for static WCET analysis with a symbolic representation of abstract pipeline models, and assess its effectiveness on a set of industrial benchmarks. We have developed a prototype implementation which is integrated into the commercial WCET analysis tool `aiT`. In our prototype implementation, we employ the model of a real-life processor, the Infineon TriCore. The model was developed and tested within `aiT`. This enables a meaningful performance comparison between the two implementations, which produce the same analysis results.

To arrive at an efficient symbolic analysis that scales to industrial-size programs, we have not only incorporated well-known optimizations from symbolic model checking but also novel domain-specific optimizations that leverage properties of the processor and the program.

### 2.7 Context Switch Costs

In preemptive real-time systems, scheduling analyses are based on the worst-case response time of tasks. This response time includes upper bounds on the execution time and context switch costs. In case of preemption, cache memories may suffer interferences between memory accesses of the preempted and of the preempting task. These interferences lead to some additional reloads that are referred to as cache-related preemption delay (CRPD). This CRPD constitutes a large part of the context switch costs.

Upper bounds on the CRPD are usually computed using the concept of useful cache blocks (UCB). These are memory blocks that may be in cache before a program point and may be reused after it. When a preemption occurs at that point the number of additional cache-misses is bounded by the number of useful cache blocks. We tighten the CRPD bound by using a modified notion of UCB: Only cache blocks that are definitely cached are considered useful by our approach [AB09].

## References


[AB09] Sebastian Altmeyer and Claire Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)*, pages 109–118. IEEE Computer Society, July 2009.

[Alt96] P. Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks*. PhD thesis, Universität Paderborn, 1996.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. pages 238–252, Los Angeles, California, 1977.

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par*, pages 1298–1307, 1997.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, 2002.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469 – 485, 2001.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[GRG09] Daniel Grund, Jan Reineke, and Gernot Gebhard. Branch target buffers: WCET analysis and timing predictability. In *15th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009*, August 2009.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture an the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.

[HR09] Jörg Herter and Jan Reineke. Making dynamic memory allocation static to support WCET analyses. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

[HRW08] 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.

[HSR+00] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *The Journal of Real-Time Systems*, pages 121–148, May 2000.

[HWH95] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. pages 288–297, December 1995.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, November 2008.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[RSW02] T. Reps, M. Sagiv, and R. Wilhelm. Shape analysis and applications. In Y N Srikant and Priti Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, pages 175 – 217. CRC Press, 2002.

[SM07a] Ingmar Stein and Florian Martin. Analysis of path exclusion at the machine code level. In *WCET*, 2007.

[SM07b] Ingmar Stein and Florian Martin. Analysis of path exclusion at the machine code level. In *Proceedings of the 7th International Workshop on Worst-Case Execution-Time Analysis*, July 2007.

[SS07] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *MEMOCODE*, pages 39–48, 2007.

[TFW00] 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, May 2000.

[The02a] Henrik Theiling. *Control Flow Graphs For Real-Time Systems Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[The02b] Henrik Theiling. Ilp-based interprocedural path analysis. In *Embedded Software (EMSOFT)*, volume 2491 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2002.

[The04] Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[TSH+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.

[Wil05] Reinhard Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.

[WLP+10] 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*. Springer-Verlag, 2010. To appear.