# Confidence in Timing

Daniel Kästner, Markus Pister, Gernot Gebbard, Marc Schlickling, Christian Ferdinand

# Confidence in Timing

Daniel Kästner, Markus Pister, Gernot Gebhard, Marc Schlickling, Christian Ferdinand

AbsInt GmbH, Science Park 1, 66123 Saarbrücken, Germany

**Abstract** All contemporary safety standards require to demonstrate the absence of functional and non-functional safety hazards. In real-time systems this includes demonstrating the absence of critical timing hazards. To meet this verification objective it is necessary to show the correctness of the timing behavior with adequate confidence. Adequate confidence means that the evidence provided can be trusted beyond reasonable doubt. There are two main sources of doubt: the logic doubt associated with the validity of the reasoning and the epistemic doubt associated with uncertainty about the underlying assumptions. A fundamental timing property is the per-task worst-case execution (WCET). It is an ingredient for determining all higher-level timing concepts like worst-case response times, and system-wide end-to-end times. This article gives an overview of the challenges in ensuring timeliness of real-time software focusing on the worst-case execution time problem. It describes the principles of abstract interpretation-based WCET analysis and summarizes the confidence argument for applying it in the certification process of safety-critical software, addressing both logic and epistemic doubt.

## 1 Introduction

Safety standards like DO-178B/DO-178C, ISO-26262, IEC-61508, or CENELEC EN-50128 require to demonstrate the functional safety of the software. Functional correctness has to be demonstrated with respect to the specified requirements and the absence of critical non-functional hazards – including timing hazards in real-time systems – has to be shown. This substantiation has to be done with adequate confidence. Adequate confidence means that the evidence provided can be trusted beyond reasonable doubt. There are two main sources of doubt: the logic doubt associated with the validity of the reasoning and the epistemic doubt associated with uncertainty about the underlying assumptions [14].

Demonstrating timing correctness requires to show that all real-time tasks meet their deadlines, or that deadline violations do not compromise the safety of the system. To demonstrate deadline adherence the worst-case response times (WCRT) of the real-time tasks in the system have to be determined. The WCRT of a task is based on its worst-case execution time (WCET) and takes additional overhead caused, e.g., by task preemptions and task blocking into account.

Due to the characteristics of modern hardware and software architectures determining the WCET of a task has become a challenge. Speculative hardware

features, timing anomalies and domino effects reduce the significance of individual timing measurements and as full test coverage cannot be achieved, no safe test end criterion is available. Also, runtime measurements typically are invasive, or require a special hardware setup. In consequence, with measurement-based techniques to determine the WCET logic and epistemic doubts usually remain.

The logic doubt can be eliminated by using formal methods for timing analysis. The theory of abstract interpretation provides a formal methodology for semantics-based static analysis of dynamic program properties. Applied to worst-case execution time analysis it allows provably sound overapproximations of the WCET to be determined even for complex architectures exhibiting timing anomalies and domino effects. Systematic timing faults caused by the interplay between software and hardware can be detected and eliminated. With the soundness of the analysis methodology established, it remains to show the correctness of the underlying hardware model, the correctness of the analyzer implementation, and to investigate the underlying assumptions about the physical world. Confidence in the microprocessor model has to be built on empirical evidence. We propose the following strategy: The analytically determined WCET bounds for representative programs or code snippets are compared with measurement data: measured times must always be below the analytically computed WCET bound. Furthermore automatic trace validation allows a cycle-accurate validation of the model down to the level of individual pipeline events and bus signals, based on the automatic processing of trace files created on the real hardware. Implementations of abstract interpretation based program analyzers can be automatically generated from the mathematical analysis specification, hence enabling high implementation quality. Qualification Support Kits enable tool users to demonstrate the correct functioning of the tool in their operational environment in an automatic way. Part of the tool qualification also is a detailed summary of all underlying assumptions of model, implementation, interfaces, and operational conditions. Qualification Software Life Cycle reports document the soundness of the tool development and validation processes. All these measures together enable the epistemic doubt to be eliminated.

## 2 Timing Predictability

Ensuring timing predictability is an active area of research which involves many different fields, ranging from predictable hardware design, determining the worst-case execution time (WCET), WCET-aware compilation, programming language support for real-time properties, to system-level scheduling and analysis. An overview of current research in all related fields can be found in [1].

In general, a system is predictable if it is possible to predict its future behavior from the information about its current state. We consider predictability under the assumption that the hardware works without unexpected errors. Hardware faults like soft errors or transient faults have to be addressed by specific error handling mechanisms to ensure overall system safety.

In [1] the program input and the hardware state in which execution begins are identified as the primary sources of uncertainty in execution time. *Hardware-*

*related timing predictability* can be expressed as the maximal variance in execution time due to different hardware states for an arbitrary but fixed input. Analogously, *software-related timing predictability* corresponds to the maximal variance in execution time due to different inputs for an arbitrary but fixed hardware state. A basic assumption is uninterrupted program execution without interferences. In a concurrent system, interferences due to concurrent execution additionally have to be taken into account.

To ensure the correct timing behavior it is necessary to demonstrate the deadline adherence of each task. To this end, the worst-case execution time of each task has to be determined, i.e. the concept of software-related predictability as defined above can be reduced to the predictability of the worst-case execution path.

At the microprocessor level for non-pipelined architectures one can simply add up the execution times of individual instructions to obtain a bound on the execution time of a basic block. Modern embedded processors try to maximize the instruction-level parallelism by sophisticated performance enhancing features. Pipelines increase performance by overlapping the executions of consecutive instructions. Hence, for timing analysis it is not sufficient any more to consider individual instructions in isolation. Instead, they have to be analyzed collectively – together with their mutual interactions – to obtain tight timing bounds.

Another important aspect is the predictability of the cache replacement policy. The Least-Recently-Used (LRU) replacement policy has the best predictability properties. Employing other policies, like Pseudo-LRU (PLRU), or First-In-First-Out (FIFO), or Random, yields less precise WCET bounds because fewer memory accesses can be precisely classified [13] and causes higher execution time variability. Furthermore, the timing analysis efficiency degrades because the analysis has to explore more possibilities. Other commonly used performance-enhancing features are static/dynamic branch prediction, speculative execution, out-of-order execution, branch history tables, or branch target instruction caches. In general, the challenges in determining worst-case execution time bounds originate from the complexity of the particular execution pipeline and the connected hardware devices. A detailed discussion of the predictability of microprocessor features, including multi-core processing, and their effect on timing analysis and measurements is given in [20].

Many of these hardware features can cause *timing anomalies* [13] which render WCET analysis and measurement more difficult. Intuitively, a timing anomaly is a situation where the local worst-case does not contribute to the global worst-case. For instance, a cache miss – the local worst-case – may result in a globally shorter execution time than a cache hit because of hardware scheduling effects. In consequence, for timing analysis it is not safe to assume that the memory access causes a cache miss; instead both states have to be taken into account.

An especially difficult class of timing anomalies are domino effects [9]: A system exhibits a *domino effect* if there are two hardware states $a, b$ s.t. the difference in execution time (of the same program starting in $a, b$ respectively) cannot be bounded by a program-independent constant factor, but may grow arbitrarily

high. E.g., given a program loop, the executions never converge to the same hardware state and the difference in execution time increases in each iteration. For timing analysis the consequence is that loops have to be analyzed very precisely and the number of machine states to track can grow high. For timing measurements this means that the difference between measured and true worst-case execution time caused by an incomplete hardware state coverage can grow arbitrarily high.

An architecture without timing anomalies and domino effects is called *fully timing compositional* [20]. On such architectures timing analysis can safely follow local worst-case paths only. Timing measurements for code snippets are compositional. One example for this class is the ARM7 reference architecture. However, most contemporary architectures exhibit timing anomalies (e.g., TriCore TC1797), or domino effects (e.g., PowerPC 755).

## 3 Measuring Time

Testing and measuring is an integral part of the cognitive scientific process. However, measuring the execution time of software faces some particular challenges. In general, timing measurements can be done at different levels, and with different purposes. This section gives an overview.

A general limitation of testing methods is their incompleteness: exhaustive testing usually is not possible. According to DO-178B/DO-178C for verification testing alone is not enough since testing cannot show the *absence* of errors. Identifying safe end-of-test criteria for "nonfunctional" program properties like worst-case execution time is an unsolved problem. End-to-end measurements aim at directly measuring the worst-case execution time of a given task. However, the worst-case path is unknown and hence cannot be explicitly stimulated. Exhaustive testing first requires to stimulate all potential execution paths. Additionally, on all but fully timing-compositional architectures different possible microprocessor states have to be taken into account. The same code snippet can have very different execution times depending on the initial hardware state of the processor at the beginning of the measurement. The worst-case hardware state for executing a code sequence usually is not known. Therefore for exhaustive testing all potential execution paths through the software have to be stimulated for all possible hardware states. The resulting search space is gigantic and cannot be exhaustively covered with acceptable effort.

Structural coverage criteria like MC/DC coverage are not applicable as they do not capture the execution paths traversed and there is no indication whether the worst-case path has been observed. It is also not possible to define a reliability metrics for timing measurements based on the time spent for measurements. There is no indication how often a specific execution path has been exercised during the observation period. In consequence for software-based systems no statistical failure rates are available which are comparable to those used for hardware components with typical requirements between $10^{-5}$ and $10^{-9}$ failures per hour of operation.

In hybrid approaches the execution times of smaller code sequences are measured and composed to determine critical paths. With timing measurements

at the basic block level there is only one path to measure, but it is often not possible to stimulate it for all potential initial hardware states with acceptable effort. The necessary tracing effort is high. Measuring larger subpaths reduces the tracing effort but incurs the danger of missing valid execution paths. Due to timing anomalies and domino effects combining measured times for different code blocks is not safe, and it is hard to determine a valid safety margin. On the other hand, combining measurements from different contexts also might lead to significant overestimations of the execution time.

Another problem dimension is how to obtain the measurement data. The simplest approach is *invasive* by adding instrumentation code to collect a time stamp or read the CPU cycle counter. The instrumentation code interferes with the timing behavior of the software under test and in general, it is not possible to separate the effect of the instrumentation code from the effect of the original code. Assessing the effect of instrumentation is a prerequisite for using invasive test methods in a safety case. As this is not possible in general, one solution is to consider the instrumentation code to be part of the production software. However this induces further problems, e.g., creating the instrumentation is subject to the same criticality level as developing the application software, and the available processor time is reduced. Mixed hardware/software instrumentation techniques enable a more lightweight instrumentation. Non-intrusive measurements are supported by logic analyzers or hardware tracing mechanisms, e.g., the IEEE-ISTO 5001-2003 (NEXUS) standard and the ETM tracing mechanism.

In consequence, with measurement-based testing, in general, it is not possible to derive *safe* bounds on the worst-case execution time. Still, they provide valuable feedback for assessing the timing behavior in soft real-time systems. They play an important role for debugging, and for determining and optimizing the average-case execution time. Testing techniques are required to address transient hardware faults, or incorrect interrupt handling. Measurement-based testing also is instrumental in validating the correctness of hardware models used for static WCET analysis (cf. Sec. 5).

## 4 Static WCET Analysis

A comprehensive survey of methods and tools for determining the worst-case execution time is given in [19]. The most successful formal method for WCET computation is Abstract Interpretation-based static program analysis. Static program analyzers compute information about the software under analysis without actually executing it. Semantics-based static analyzers use an explicit (or implicit) program semantics that is a formal (or informal) model of the program executions in all possible or a set of possible execution environments. Most interesting program properties – including the WCET – are undecidable in the concrete semantics. The theory of abstract interpretation [2] provides a formal methodology for semantics-based static analysis of dynamic program properties where the concrete semantics is mapped to a simpler abstract model, the so-called abstract semantics. The static analysis is computed with respect to that abstract semantics, enabling a trade-off between efficiency and precision.

A static analyzer is called *sound* if the computed results hold for any possible program execution. Applied to WCET analysis, soundness means that the WCET bounds will never be exceeded by any possible program execution. Abstract interpretation supports formal soundness proofs for the specified program analysis.

Like model checking and theorem proving, abstract interpretation is recognized as a formal method by the DO-178C and other safety standards (cf. DO-333, Formal Methods Supplement to DO-178C). It is based on a mathematically rigorous concept and provides the highest possible confidence in the correctness of the results (cf. IEC-61508, Ed. 2.0, Table C.18).

Over the last few years, a more or less standard architecture for timing analysis tools has emerged [4,7] which is composed of three major building blocks:

- control flow reconstruction and static analyses for control and data flow,
- micro-architectural analysis, computing upper bounds on execution times of basic blocks,
- path analysis, computing the longest execution paths through the whole program.

The core of the analysis is the micro-architectural analysis where basic block timings are determined using an abstract processor model (*timing model*) to analyze how instructions pass through the pipeline taking cache-hit or cache-miss information into account.

In the following sections we will focus on the commercially available tool aiT which implements the architecture described above. The tool has been successfully employed in the avionics [7,17] and automotive [11] industries to determine precise bounds on execution times of safety-critical software. It is available for a variety of microprocessors ranging from simple 16-bit processors like ARM7 to complex superscalar processors with timing anomalies and domino effects like Freescale MPC755, or MPC7448.

## 5 Providing Confidence

In general confidence about a system property is based on providing evidence that the conditions of the physical world have been appropriately taken into account and that the reasoning about the correctness of the system reaction is sound. A fundamental system property is the per-task worst-case execution (WCET) from which worst-case response times and end-to-end times can be derived. The corresponding correctness requirement is to show that all hard real-time tasks definitely meet their deadline. In general it is possible to provide for mechanisms that detect timing violations and initiate mitigating actions, e.g., based on deadline monitoring or runtime monitoring. Possible reactions to detections of timing violations include activation of redundant system elements, mode switches to degraded mode operation, or entering the safe state. Whereas this is acceptable to address hardware faults like soft errors or transient faults it is problematic for software errors as this incurs the danger of unacceptable degradation of availability and can affect system safety. The vast majority of software errors are systematic errors and in general no statistical failure rates

for software are available. Therefore it is preferable to demonstrate stringent deadline adherence under the assumption of error-free hardware behavior and use fault mitigation for hardware errors which are amenable to statistical failure rates.

The worst-case execution time is influenced by the semantics of the software under analysis, the properties of the microprocessor used, and finally the system-level properties and its connection to the physical world. To establish confidence in reasoning about worst-case execution time it has to be shown that all these factors have been correctly taken into account and that the means to derive the worst-case execution time from them are technically correct and complete.

The input to static WCET analyzers is fully linked binary machine code. Binary machine code is a programming language that provides a formal description of the behavior of the programs on the underlying hardware architecture. Using abstract interpretation a formal abstract semantics is defined as the basis of the static program analysis to be performed that focuses on the worst-case timing behavior. This abstract timing semantics is based on a formal model of the microprocessor used. By design, Abstract Interpretation based static analyzers provide full control and data coverage. As detailed above abstract interpretation supports formal correctness proofs: it can be proven that an analysis will terminate and that it computes an overapproximation of the concrete semantics, i.e., that the analysis results are *sound*: the true worst-case execution time will never be underestimated. There is a variety of scientific publications about the theory of Abstract Interpretation, e.g., [2]. The properties of aiT WCET Analyzer are discussed in [17,6,8], soundness and correctness proofs have been published, e.g., in [5,18,3].

In consequence, abstract interpretation as a formal verification method enables the analysis design to be proven correct. This eliminates the logic doubt: based on formal specifications of the input program, i.e. the binary executable and the microprocessor, the specified analysis will compute sound results. It remains to be shown that the microprocessor model correctly models the physical hardware, that the implementation of the static analyzer correctly realizes the mathematical specification, and that the properties of the surrounding physical world are correctly taken into account. In the following we will address each of these topics in turn.

### 5.1 Confidence in Model

Modern processors and peripheral components like memory controllers are automatically synthesized from formal design specifications, mostly in VHDL or Verilog. If a VHDL/Verilog model is available an abstract timing model can be semi-automatically derived from the design specification as described in [12,15,16]. The model derivation process ensures the correctness of the timing model as both the synthesized hardware and the corresponding timing model have been created from the same formal specification.

Unfortunately, processor manufacturers often do not provide access to their formal hardware designs for confidentiality reasons. Then the timing model has to be constructed manually by human experts based on the available system

documentation. As such documentation might contain errors and/or does not describe all details of the instruction flow through the processor pipeline, a subsequent model validation is necessary to demonstrate correctness.

A first step to get confidence in the timing model is to compare analytically computed WCET bounds for representative programs or code snippets with measured times. No analytically computed WCET bound must ever underestimate any measured execution time, i.e., the bounds must always be greater than or equal to the measured execution time. However, this only gives a coarse-grained view on the quality of the timing model because local underestimations might be shadowed by overestimations in other code parts so that potential errors might remain undetected. Therefore, additional validation activities are necessary.

Applying the timing model to a program not only yields a WCET bound but also results in a prediction of all possible execution paths through the program including all possible microprocessor states during the execution. In consequence the model can also be validated by comparing the prediction with the observable events of the corresponding concrete execution of the program on the hardware, provided that the observed events are captured by the model. Depending on the underlying hardware and measurement setup, events can be traced at different levels of granularity: performance counter values, active bus signals, instructions executed, and routines (C functions) executed.

Some processors feature so-called *performance monitoring facilities* to monitor and count predefined events such as processor clock ticks, cache misses, dispatched instructions, or mispredicted branches. Performance monitoring counters provide very fine-grained insights in the processor core for a certain amount of clock ticks. Each of the performance counter registers can be configured to count several events. To understand the insights of a processor core, the traced time period must be chosen very carefully, and measurements need to be repeated several times to allow for tracing different events.

On many architectures it is possible to trace the communication of the processor with its "outer world", i.e., to monitor the change of signal values on the system bus. The resulting *bus trace* provides insights on the requests generated by the processor core and thus allows to draw conclusion on the branch prediction behavior and other advance pipeline features.

Performance monitoring and bus traces complement each other: performance monitoring provides insight in the internal behavior of the processor core, and bus traces allow the interaction of the core with its peripherals to be observed.

A less fine-grained method for investigating the behavior of an embedded architecture is to evaluate *instruction traces* as, e.g., provided by NEXUS traces. Every instruction emits an event at the beginning of its execution, and one event at the exit of its execution. The ordering of event occurrences provides insights on the executed program paths and furthermore allows to investigate pipelined execution capabilities of the underlying architecture.

Coarse-grained measurement methods evaluate the execution behavior at the level of *routine execution* (i.e. at the C function level). Different execution contexts can be distinguished by reference to the call history. However, it is not

possible to gain precise information about the control flow inside routines. Such measurement methods do not allow to infer detailed information about the processor behavior.

The aiT timing model can be extended to predict any of the above mentioned event types. The analysis does not predict a single trace of events. Instead the result is a *prediction graph* that – due to the underlying abstract interpretation principles – describes a sound overapproximation of all possible traces of events that are observed in reality. Hence, any measured event trace has to be part of this prediction graph. Therefore validating measured traces of events against the statically obtained prediction graph provides evidence of timing model correctness. This trace validation process can be fully automated and thus allows for an industry-strength timing model validation.

Our trace validation methodology comprises the following steps. First the execution behavior is measured on the real hardware[1]. Second the prediction graph is obtained from aiT. Finally, a graph search determines whether the measured trace of events is contained in the prediction graph. The trace validation is successful if there exists a path that comprises the events in exactly the same order in which they have been observed.

The entire validation process can be fully automated. In this fashion aiT has been successfully validated even for very large amounts of measurement data. The available measurement typically comprise several millions of trace lines. Table 1 shows an excerpt of the trace validation data that was used to validate aiT for several architectures.

| Architecture | Application Type | Binary Size | Event Types | Trace Lines |
|---|---|---|---|---|
| M68020 | Avionic | 14MB | bus | 4 232 000 |
| MPC5xx | Avionic | 3MB | instr | 3 879 000 |
| MPC755 | Avionic | 120MB | bus | 9 468 000 |

**Table 1.** Excerpt of industrial trace data used for aiT tool validation.

### 5.2 Confidence in Implementation Quality

The concise mathematical specification of program analyses provided by the theory of abstract interpretation allows implementations of program analyzers to be automatically generated. The Program Analyzer Generator PAG automatically generates implementations of static program analyzers from a formal specification of an abstract interpretation problem [10]. In order to generate an analyzer using PAG, two specifications have to be written: one for the definition of the data structures, including the abstract domain, and one containing the analysis parameters, definition of the abstract transfer functions and, optionally, some support functions. The generative approach reduces the risk of implementation errors and frees the PAG user from having to implement the domain functionality, the traversal of the control flow graph, and suitable fixpoint algorithms. Minimizing manual coding eliminates a major source of errors.

PAG is freely available for teaching and research[2] and has been used in numerous research and teaching projects. Furthermore it has been used to generate

[1] It is also possible to use a behavioral VHDL model to observe the hardware behavior.
[2] http://www.program-analysis.com/

more than 40 commercial static analyzers at AbsInt, including aiT WCET Analyzers and static stack usage analyzers for various microcontrollers.

### 5.3 Tool Qualification & Qualification Software Life Cycle Reports

To provide high confidence in the correct functioning of a tool it is necessary to demonstrate that the tool works correctly in the operational context of its users. This is a common requirement of most current safety standards. The correct functioning of a tool might be affected by the OS version, system libraries installed, software patch levels, etc. Moreover, depending on the user's development process structure and tool landscape the probability for detecting tool errors may vary. Therefore taking into account the operational context of tool usage is essential for tool qualification.

From the perspective of a tool user, qualifying a software tool causes considerable effort. The functional requirements of the tool have to be specified, a test plan has to be developed, tests have to be executed and documented. Moreover the qualification effort has to be repeated for each development project to be certified. This makes it very desirable to do automatize the tool qualification process. Such an automatic tool qualification can be done by dedicated *Qualification Support Kits (QSKs)* as shipped as a part of a software tool.

In the following we give an overview of a QSK for aiT whose structure is representative for general qualification support kits. It is centered around an automatic validation suite and essentially consists of two parts, a report package and a test package.

The *report part* consists of two different documents: the *Tool Operational Requirements (TOR)* and the *Verification Test Plan (VTP)*. The TOR lists the tool functions and technical features which are stated as low-level requirements to the tool behavior under normal operating conditions. Additionally, the TOR describes the tool operational context and conditions in which the tool computes valid results. Hence this part specifically focuses on the *epistemic doubt*. It summarizes all basic assumptions about the physical world that have to be satisfied for correct tool operation. This includes: unsupported microprocessor options or configurations, system parameters (e.g., scheduling strategy, occurrence of DMA, dynamic RAM refreshes, or exceptions), etc. The VTP defines the test cases demonstrating the correct functioning of all specified requirements from the TOR. Test case definitions include the overall test setup as well as a detailed structural and functional description of each test case. The *test part* contains an extensible set of test cases with a scripting system to automatically execute them and generate reports about the results.

Depending on the safety standard and the criticality level of the application providing additional confidence about the tool software development processes may be required. The AbsInt *Qualification Support Life Cycle Data (QSLCD)* reports contain documents which detail the tool development processes, e.g., the software development plan, the quality plan, quality assurance records, the software verification plan and software verification results.

## 6 Conclusion

All current safety standards consider the correct timing behavior as a part of the functional safety of real-time systems. A fundamental timing property is the per-task worst-case execution (WCET) from which all higher-level timing concepts like worst-case response times, and system-wide end-to-end times are derived. Determining the WCET from measurement-based testing techniques is subject to logic and epistemic doubts. Speculative hardware features, timing anomalies and domino effects reduce the significance of individual timing measurements and as full test coverage cannot be achieved, no safe test end criterion is available. Also, runtime measurements typically are invasive, or require a special hardware setup.

The theory of abstract interpretation provides a formal methodology for semantics-based static analysis of dynamic program properties. Applied to worst-case execution time analysis it allows provably sound overapproximations of the WCET to be determined even for complex architectures exhibiting timing anomalies and domino effects. The analysis is based on a formal model of the underlying microprocessor. With the soundness of the analysis methodology established, it remains to show the correctness of the underlying hardware model, the correctness of the analyzer implementation, and to investigate the underlying assumptions about the physical world. We have developed a strategy to address all these issues based on a combination of formal proofs, careful documentation and automatic trace-based measurements. The goal of these measurements is to provide empirical evidence on the physical hardware that modeling and implementation is correct. With all these measures together both the logic and the epistemic doubts can be addressed.

## Acknowledgement

The work presented in this paper has been supported by the European FP7 project T-CREST, and the EU ARTEMIS Joint Undertaking under grant agreement no. 269335 with the German BMBF (MBAT project).

## References

1. P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 2013. Accepted.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.
3. C. Cullmann. *Cache Persistence Analysis for Embedded Real-Time Systems.* PhD thesis, Universität des Saarlandes, 2013. To be published.
4. A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis.* Phd thesis, Uppsala University, 2003.
5. C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.

6. C. Ferdinand and R. Heckmann. Worst-case execution time – a tool provider's perspective. In *Proceedings of the International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 340–345, Orlando, USA, May 2008. IEEE Computer Society.
7. 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 *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *LNCS*, pages 469–485. Springer, 2001.
8. D. Kästner and C. Ferdinand. Efficient Verification of Non-Functional Safety Properties by Abstract Interpretation: Timing, Stack Consumption, and Absence of Runtime Errors. In *Proceedings of the 29th International System Safety Conference ISSC2011*, Las Vegas, 2011.
9. T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium (RTSS)*, December 1999.
10. F. Martin. PAG – an efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.
11. NASA Engineering and Safety Center. Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation, 2011.
12. M. Pister. *Timing Model Derivation – Pipeline Analyzer Generation from Hardware Description Languages.* PhD thesis, Saarland University, October 2012.
13. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In F. Mueller, editor, *International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2006.
14. J. Rushby. Logic and Epistemology in Assurance Cases. In D. Cofer, J. Hatcliff, M. Huhn, and M. Lawford, editors, *Software Certification: Methods and Tools (Dagstuhl Seminar 13051)*, volume 3, pages 111–148, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
15. M. Schlickling. *Timing Model Derivation – Static Analysis of Hardware Description Languages.* PhD thesis, Saarland University, January 2013.
16. M. Schlickling and M. Pister. Semi-automatic derivation of timing models for wcet analysis. In J. Lee and B. R. Childers, editors, *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 67–76, Stockholm, Sweden, April 2010. Association for Computing Machinery (ACM).
17. J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.
18. S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD thesis, Universität des Saarlandes, 2004.
19. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.
20. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, July 2009.