WCET and Mixed-criticality: How to Quantify the Confidence in WCET Estimations?

Sebastian Altmeyer, Björn Lisper, Claire Maiza, Jan Reineke, Christine Rochange

WCET and Mixed-Criticality: What does Confidence in WCET Estimations Depend Upon?*

Sebastian Altmeyer[1], Björn Lisper[2], Claire Maiza[3], Jan Reineke[4], and Christine Rochange[5]

1 **University of Luxembourg, Luxembourg**
`sebastian.altmeyer@uni.lu`
2 **Mälardalen University, Sweden**
`bjorn.lisper@mdh.se`
3 **Université Grenoble Alpes, Verimag, France**
`claire.maiza@imag.fr`
4 **Saarland University, Germany**
`reineke@cs.uni-saarland.de`
5 **University of Toulouse, France**
`rochange@irit.fr`

Abstract

Mixed-criticality systems integrate components of different criticality. Different criticality levels require different levels of confidence in the correct behavior of a component. One aspect of correctness is timing.

Confidence in worst-case execution time (WCET) estimates depends on the process by which they have been obtained. A somewhat naive view is that static WCET analyses determines safe bounds in which we can have absolute confidence, while measurement-based approaches are inherently unreliable. In this paper, we refine this view by exploring sources of doubt in the correctness of both static and measurement-based WCET analysis.

1998 ACM Subject Classification C.3 Computer Systems Organization – Real-Time and Embedded Systems

Keywords and phrases mixed criticality, WCET analysis, confidence in WCET estimates

Digital Object Identifier 10.4230/OASIcs.WCET.2015.65

1 Introduction

Due the integration of multiple safety levels (A to E in DO178B for certification in avionics, or A to D in ISO 26262, a functional safety standard for automotive), tasks of different criticality may be executed on a shared platform. A naive approach to certifying such a *mixed-criticality* system is to apply the certification methods corresponding to the highest present level of criticality to *all* tasks. The drawback of this approach is that low-criticality tasks become unnecessarily costly to validate and the system analysis potentially pessimistic.

Research in mixed-criticality scheduling targets the validation of these system assuming the certification requirements are reflected in the execution time bounds, which vary depending on the associated criticality. The original model by Vestal [35] suggests two levels of timing estimations:

* This work was partially supported by French ANR under grant ANR-12-INSE-0001, the German Science Foundation (DFG) as part of the Transregional Collaborative Research Center SFB/TR 14 (AVACS) and by the COST Action IC1202: Timing Analysis On Code-Level (TACLe).

- C(HI) is a high-confidence WCET estimate, and
- C(LO) is a lower confidence estimate of the WCET,

but there may be as many WCET estimations as safety levels.

The rationale behind this model is to guarantee schedulability of all tasks with their C(LO) bounds, while ensuring that if a task exceeds its low criticality execution time bound, the highly-critical tasks are able complete their execution within their deadlines as long as their execution times do not exceed their C(HI) bounds. Intuitively, C(LO) $\leq$ C(HI) as it is assumed in Vestal's model. However, a more reliable WCET analysis provides higher confidence in the validity of its estimation, but does not necessarily result in a greater bound. Also, increasing the effort to analyse a task does not necessarily increase the bound, but may even reduce it.

In this paper, we present our point of view on the sources of these different levels of confidence in the WCET estimations. Our aim is to discuss the sources of doubt in the correctness of WCET estimations. In contrast to [10], where confidence in WCET estimations and monotonicity of WCET estimations with respect to the different certification levels have already been shortly discussed, we highlight the problem from the perspective of timing analysis. We first discuss confidence in static and measurement-based WCET analysis methods, then focus on the impact of multi-core platforms on these sources of doubts and specifically on the interferences on shared resources. We conclude with a discussion of open problems.

Please note that we restrict our attention in this paper to deterministic approaches currently used in industry (due to limited space). As future work, we plan to enlarge our study to recently introduced probabilistic approaches.

2 On Confidence in Static WCET Analysis

2.1 Structure of Static WCET Analysis Tools

Static timing analyses compute bounds on a task's execution time by analysing the task characteristics and determining its behaviour on the target machine statically, i.e, without executing the task on the target platform. The techniques employed by static timing analyses, such as abstract interpretation or model checking, are borrowed from the related fields compiler construction and verification and are meant to be sound by construction.

A static timing analysis typically consists of three phases, ISA-level analysis, microarchitectural analysis and path analysis. ISA-level analysis derives flow information of the task, such as loop bounds, effective memory addresses of memory read or write and pointer addresses. Microarchitectural analysis derives bounds on the execution times of each basic block. Path analysis combines the information of the previous steps and determines the longest execution path through the program.

Microarchitectural analysis has to resort to the level of the binary as only on this level complete information about the task behaviour is available. ISA-level analysis can operate on both, the high-level and on the binary, yet requires a control-flow graph representation of the program. Consequently, additional steps are needed to bridge the gap between the different representations and to establish a connection between the main phases of the analysis.

2.2 ISA-level Analysis

Some important supporting analyses depend only on the untimed, "functional" semantics of the code. This includes *program flow analysis*, which attempts to find program flow

constraints such as loop bounds, or infeasible path constraints. Also a conventional *value analysis* is often needed, for purposes like bounding the possible addresses for memory accesses.

There are several sources of uncertainty regarding these analyses. We focus mainly on three points: uncertainty due to user annotations, uncertainty due to the analysis method, and uncertainty due to the traceability of information from source level to binary level.

One concern are the assumptions that often have to be made about the environment in which the code runs. Analysis tools typically allow the user to specify properties that can affect the outcome of the analysis, like limitations on value ranges for inputs, or whether some variables should be considered volatile. There is always a risk that such manually-specified properties are false.

Another potential source of uncertainty is if the analysis is indeed unsound. A value- or program-flow analysis must always rely on some assumptions on the semantics of the code: if there are situations where these are not fulfilled, then value ranges or program flows may be underestimated which in turn can yield an unsafe WCET estimate. For instance it is not uncommon that analyses consider numbers to be unbounded, "mathematical" numbers when indeed they have a finite representation in the software. Fig. 1 shows an example where a loop bounds analysis that rests on this assumption will fail. Such an analysis will find that the loop body can be executed only one time, whereas in reality `i`, which is an 8-bit unsigned number, will wrap around from 254 to 0 when incremented by 2 causing a non-terminating loop.

Analyses that involve floating-point numbers can suffer from unsoundness since it may be hard for a tool to support all the varieties of floating-point arithmetics that different processors use. It is therefore common that analysis tools use some standard floating-point arithmetics such as IEEE floating-point arithmetics, or the native arithmetics of the computer where the tool executes. However, this may not be the arithmetics used by the target machine, which then can yield an unsound analysis.

```
unsigned char i;
i = 254;
while (i <= 255) do {
    i = i + 2;
}
```

Figure 1 A simple example of a code with wrap-around.

In a similar fashion, if low-level code is analysed, the analysis can become unsound if it assumes the wrong endianness of the target architecture.

A final potential source of unsound analysis are pointers. If the analysis cannot bound a pointer in a program point where the pointer yields the address for a write, then a sound analysis must assume (very pessimistically) that the write may occur anywhere in the memory (possibly including, for instance, the program code). Thus, after such a write, basically all information about what may happen next in the program is lost. It is therefore common that analyses assume that writes using such unbounded pointers cannot be more than "reasonably" out of bounds: for instance, it is commonly assumed that they cannot modify the program code.

For maximal confidence, value- and program flow analyses must be performed on the linked binary code. However, analyses can be hard to perform on this level due to lack of information about types, syntactic structure, etc. Therefore it is not uncommon that these analyses are attempted at the source code level instead, with the results subsequently being mapped to the binary level with the aid of debug information or alike. However, even the results of an analysis that is sound on the source level may not be sound for the compiled binary due to compiler optimisations changing the structure of the code. Even if optimisations are turned off, some compilers may still perform code transformations like

turning `while-do` loops into `do-while` loops. Work has been done how to trace program flow constraints through compiler optimisations [16, 24], but production compilers do not implement these solutions.

2.3 Microarchitectural Timing Models

The low-level analysis step computes the worst-case execution times of code fragments, such as basic blocks. It is based on a cycle-accurate model of the target platform which specifies the hardware behavior when executing a sequence of instructions. There exist several ways to build such a model:

- The hardware timing model can be specified by the tool designer or by the end-user from the processor manual that is usually publicly available from the processor manufacturer. As mentioned in [27], this task is both time-consuming and error-prone due to: (a) missing or even incorrect documentation (user manuals generally focus on specifying the programming models but do not provide a detailed view of the processor internals nor accurate instruction timings), and (b) human errors when translating the natural-language description of the processor architecture given by the manual into a formal model. The reliability of such hand-crafted timing models is difficult to assess and this is even more true when the processor features complex hardware mechanisms, which are usually poorly documented.
- Measurement techniques can also be used to reverse engineer hardware parameters. In [14], monitoring registers are used while running specifically-designed micro-benchmarks to identify the processor's write and cache replacement policies. Similar techniques are used in [5] to investigate translation look-aside buffers (TLBs). New variants of the pseudo-LRU replacement policy implemented in the Intel Atom D525, the Intel Core 2 Duo E6750, and the Intel Core 2 Duo 8400 but not publicly documented could be discovered by application of automata learning [1] in case of of the Intel Atom and a combination of automatic measurements and human insight [2] in the other two cases. In [36], micro-benchmarking is used to discover the behavior of various components of an Nvidia GPU architecture, such as the warp scheduling policy. Note that all these approaches need manual work to (a) design micro-benchmarks that can exhibit hardware parameters, which might be particularly difficult in the presence of a totally original scheme that would not be described in the literature, and (b) interpret the results to determine how the processor or memory hierarchy works. In that sense, such techniques cannot provide fully reliable models but they can confirm, deny or complement the processor's description provided in the manual. In the first case, confidence in the model is increased.
- The timing model can be derived (semi-)automatically from a formal description of the hardware in a hardware description language, i.e., the microarchitecture's VHDL or Verilog model [28]. This hardware description contains the complete information required to build the microarchitectural timing model for timing analysis. Then, cumbersome and often error-prone reverse engineering is not required. Correctness of the timing model relative to the VHDL or Verilog model can be achieved and shown with comparably little effort. Due to the complexity of the hardware description and the various abstraction levels used to describe the microarchitecture, a completely automatic derivation is not possible. The timing model must be tight so as not to inflate the complexity of the microarchitectural analysis. The main obstacle remains the availability of the hardware description. Processor manufacturers are very reluctant to provide detailed hardware descriptions out of fear of plagiarism.

2.4 Microarchitectural Analysis

Given a microarchitectural timing model and a program, the task of microarchitectural analysis is to determine bounds on the execution times of program fragments. The main challenge for modern processors is that execution times of individual instructions strongly depend on the state of the microarchitecture. As an example, a memory instruction that causes a cache miss may easily take 100 times as long as one that causes a cache hit.

To correctly estimate the execution time of a program fragment, microarchitectural analysis thus needs to determine the set of states that the microarchitecture can be in when executing the program fragment. To do so microarchitectural analysis needs to take into account all possible program executions and initial states that may lead to a program fragment. To cope with the potentially very large number of cases, *abstraction* is employed where possible. Precise and efficient abstractions have been found for caches [8], whereas less structured components such as pipelines are mostly analyzed concretely, often leading to a very large number of states to be explored. Such analyses can be proven correct relative to a concrete model using the theory of Abstract Interpretation [7]. Such proofs have been carried out in paper and pencil proofs for caches and branch target buffers. Due to the lack of "strong" abstractions, beyond abstracting register and memory values, such proofs have been omitted for pipeline analyses.

To counter state-space explosion in microarchitectural analysis, it is tempting to only consider the local worst cases. Due to *timing anomalies* [20, 25], however, this is generally unsafe. It is an open problem to prove freedom of timing anomalies for models of realistic microarchitectures, while some success has been achieved in simplified scenarios [26].

Another approach to reduce analysis cost and possibly even improve precision is to analyze different components separately. For instance, one might attempt to analyze the pipeline separately from the cache. In the case of multicores a common approach is to separate the analysis of the bus blocking from the WCET analysis. Such approaches assume *timing compositionality* [11], i.e., that execution time can be safely decomposed into contributions from different components. As is the case with timing anomalies, some microarchitectures are conjectured to be timing-compositional [38], but none has formally been proven so.

A number of projects have focused on designs of or design principles for *timing-predictable* microarchitectures. This includes CompSOC [12], JOP [29], MERASA [33], Predator [38], and PRET [19]. Timing models for microarchitectures developed based on the principles identified in these projects are often simpler than those for commercial microarchitectures. This enables more precise and efficient analysis, and it also increases confidence in their correctness.

2.5 Path Analysis

Path analysis is the final step in WCET analysis. In this step, the results of microarchitectural analysis and ISA-level analysis are combined to reason about all possible program executions and their timing. Often, a program's control-flow graph (CFG) is used to bridge the gap between the two analysis levels:

- Microarchitectural analysis delivers bounds on the execution times of program fragments like the basic blocks of the CFG.
- ISA-level analysis delivers constraints on the possible paths that can be taken through the CFG, such as loop bounds.

The goal of path analysis is then to identify the worst-case execution path given the constraints obtained by ISA-level and microarchitectural analysis. Instead of explicitly exploring all

paths, state-of-the-art WCET analyzers rely on an *Implicit Path Enumeration Technique* (IPET). Possible paths and their execution time bounds are expressed as the solutions of a set of integer linear constraints. The solution maximizing the execution time can then be found by an integer-linear programming (ILP) solver. Other path analysis approaches that have been considered are based upon SAT modulo theory or model checking.

The main source of doubt in path analysis comes from the fact that execution time is estimated in numbers of machine cycles. This is an integer value that is estimated by solvers using finite number representations. Some research verified the solution for LP and/or SMT solvers: the main idea is to verify the certificate corresponding to the optimum [9, 4]. As far as we know, these studies have not yet been extended to ILP: such a verification appears possible, but the problem to solve is larger.

2.6 Confidence in Tool Implementations

A potential source of uncertainty, which is common to more or less all analysis stages, is the possibility of bugs in the tool implementations. One way to reduce the uncertainty stemming from this is to make a formal verification of the algorithm, or the code of the implementation, using a proof assistant. By reducing the *trusted code base*, i.e., the part of the code that is not verified and thus has to be trusted, confidence can be gained.

One such effort has been done in the context of the CompCert certified compiler. Maroneze [22] developed a static WCET analysis tool, using previously known techniques, in the CompCert environment and provided a formal proof of correctness for those parts of this tool that correspond to ISA-level analysis and path analysis. Correctness proofs for the microarchitectural analysis were left as future work. This work demonstrates that this kind of formal verification is within reach also for complex WCET analysis tools.

3 On Confidence in Measurement-based WCET Analysis

Measuring a task's execution time is an alternative approach towards timing verification. It provides a simple and straightforward method to derive execution time estimates. Besides the simplicity of the measurement-based approach – which is in contrast to static timing analysis – no microarchitectural model is required. Measurements can be derived for the actual binary running directly on the target architecture, thus eliminating a prominent source of uncertainty. The very same hardware used in the embedded system can also be used for the measurement.

A fundamental drawback reduces the overall confidence in measurement-based approaches: It is practically infeasible to obtain measurements that cover the complete input space and the complete set of initial processor states, let alone interferences occurring during run-time. Exhaustive measurements are simply not possible for realistically-sized tasks and modern microarchitectures.

3.1 End-to-end Measurements

End-to-end measurements represent the most naive approach towards measurement-based timing verification. The execution time from task dispatch to completion is measured for a set of program inputs and initial processor states, and based on the highest measured execution times, WCET bounds are derived. These bounds are then multiplied by a safety margin to account for potential optimism in the measurements. This safety margin was

the original motivation for mixed-criticality systems. A higher safety margin increases the WCET bound and so, also the confidence in it.

Path coverage techniques [39] are traditionally used to increase confidence in end-to-end measurements by automatically generating the set of test-cases. The automatically generated test vector is claimed to either cover all paths, or to cover at least the worst-case path. As these methods only treat the task input and not the initial processor states, doubt remains.

3.2 Hybrid Approach

Hybrid approaches [37, 17, 18, 31] use measurements to obtain estimates of the worst-case timing of program fragments. These are then combined during path analysis as in static WCET analysis tools to obtain an estimate of the WCET. There are two main approaches to obtain the timing of program fragments by measurements:

- By instrumenting the program code to obtain timestamps before and after executing each program fragment [37].
- By performing end-to-end measurements through different program paths, from which the execution time of each fragment can then be estimated [17, 18, 31].

Both approaches require the generation of inputs that drive execution through a given program point. Any unreachable code needs to be proven to be so, which may be difficult for deeply nested code.

The first approach may not deliver faithful execution-time estimates on pipelined processors, because timing is distorted through instrumentation. A too fine-grained instrumentation may also be impossible due to the large amount of trace data that must be captured at high pace. The second approach avoids these problems since it identifies the fine-grained timing models from end-to-end measurements. The approach in [18] works also for complex architectures, and can identify timing models with context-dependent costs for better precision. Both approaches may underestimate the WCET whenever execution times of program fragments depend on input data values or on the execution history, as the measurements will usually only cover strict subsets of the possible cases. On modern processors with deep pipelines, branch predictors, and caches this is the case. Exceptions are highly timing-predictable microarchitectures such as PRET machines [19].

While hybrid approaches are not guaranteed to be sound, and it is hard to quantify confidence in its results, they can also be pessimistic. The path analysis phase may combine observed worst-case timings of program fragments which may not occur together during a single program execution, which may be avoided in static analysis [32].

4 Beyond WCET Analysis – The Impact of Interference on Shared Resources

The WCET analysis exposed in previous sections considers a task that runs in isolation on the platform: its execution time is assumed not to be impacted by any external source. In practice, this assumption is rarely true: the task might be interrupted, or preempted by the scheduler for the benefit of a concurrent task, and the hardware state (e.g. cache contents) might be changed by the interrupt service routine or the preempting task; it may also be delayed by a hardware-level operation (e.g. a DMA transfer) or a task running on another core (in a multicore platform) that compete for shared resources (bus, memory, etc.). These last years, several approaches have been proposed to account for such interferences.

Techniques to estimate the cache-related preemption delay (CRPD), i.e. the number of additional cache misses due to context switching or periodic/sporadic interrupts, have received much attention recently [3, 6]. Most of the approaches use static code analysis

techniques and suffer the same confidence issues as static WCET analysis: the model and/or its implementation might be flawed.

Multithreaded/multicore platforms raise additional issues: a task can experience increased latencies upon accesses to shared resources, due to conflicts with simultaneously running tasks; and the contents of shared storage resources, such as shared L2 caches, can be evicted by co-running tasks all along the task's execution, not only at preemption points. Evicted cache contents may result in additional delays to reload information that is still in use by the task. To estimate additional latencies due to interferences from other tasks, two strategies are possible: the *blind* approach assumes the worst possible co-running task set and considers absolute worst-case latencies [23, 15]; the *scheduling-aware* approach restricts the analysis of possible conflicts to the set of tasks that can effectively run together with the task under analysis [13, 21]. Both approaches require that the sharing control policy allow upper bounding delays; this is the case for a round-robin bus arbiter, for example. In the same way as for single-core architectures, the documentation of COTS multicores might not provide enough guarantees in the sharing scheme description to be fully confident in estimated delays. For this reason, designs for time-predictable multicores have been developed in recent projects, such as T-CREST [30] or parMERASA [34]. Note that a common assumption for most of the works on this topic is that the system features timing compositionality [11], which allows analysing each component separately. Unfortunately, this property is not easy to prove.

5 Discussion and Open questions

We have discussed possible sources of errors in different WCET analysis methods: static, measurement-based, and hybrid methods, and how the potential for such errors will affect the confidence in the result. The motivation comes from the need to quantify the confidence in WCET estimates for safety-critical systems, e.g., to select C(HI) and C(LO) in Vestal's model for scheduling of mixed-criticality systems.

All WCET analysis methods have potential sources of errors. For methods like static and probabilistic analysis, which rely on mathematical models, the risk that the models do not comply with reality must be taken into account. For methods that include measurements an additional source of uncertainty is the quality of the test vectors, and the ability of the method to find inputs provoking the longest execution traces with the highest instruction execution times.

Identifying sources of reduced confidence in WCET estimates is not difficult. *Quantifying* the confidence is much harder, and we do not attempt to make any detailed assessment of the different techniques in this regard. Having said that, we do believe that static analysis methods have an edge as regards the potential to obtain high confidence in that (1) given that the underlying models are correct, the methods are provably safe, and (2) since the methods are not based on measured data, confidence can be obtained by a thorough validation of the models against the real systems, and by verifying the correctness of the algorithms and tool implementations that build on the models. The latter can be done either by a formal verification, or by a run-time verification where checking of correctness certificates is integrated in the tools.

For methods that rely on measurements, confidence also rests on the quality of the test data. Better criteria are needed to assess this quality with respect to timing. Traditional coverage criteria, like path coverage, do not consider hardware effects on timing: we would need more refined coverage criteria that take hardware states, like cache contents, into account.

However, although hard, quantified confidence in WCET analysis methods is essential if models like Vestal's model are to be applied in the design of mixed-criticality systems. Our aim is to start a discussion on how this can be done.

Acknowledgements. This paper has been initiated in the Dagstuhl Seminar 15121 – Mixed Criticality on Multicore/Manycore Platforms.

References

1 Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

2 Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

3 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

4 Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, 2011.

5 Vlastimil Babka and Petr Tuma. Investigating cache parameters of x86 family processors. In *Computer Performance Evaluation and Benchmarking*. Springer, 2009.

6 Lee Kee Chong et al. Integrated timing analysis of application and operating systems code. In *34th Real-Time Systems Symposium (RTSS)*, pages 128–139. IEEE, 2013.

7 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*, Los Angeles, January 1977.

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

9 Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *20th International Symposium on Static Analysis (SAS)*, pages 345–365, 2013.

10 Patrick Graydon and Iain Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *Workshop on Mixed Criticality Systems*, 2013.

11 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015.

12 Andreas Hansson et al. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM TODAES*, 14(1):1–24, 2009.

13 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Real-Time Systems Symposium (RTSS)*, 2009.

14 Tobias John and Robert Baumgartl. Exact cache characterization by experimental parameter extraction. In *Int'l Conf. on Real-Time and Network Systems (RTNS)*, 2007.

15 Timon Kelter et al. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2), 2014.

16 Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In *22nd International Conference on Real-Time Networks and Systems (RTNS)*, page 97, 2014.

17 Markus Lindgren, Hans Hansson, and Henrik Thane. Using measurements to derive the worst-case execution time. In *Int'l Conf. on Real-Time Computing Systems and Applications (RCTSA)*, 2000.

18 Björn Lisper and Marcelo Santos. Model identification for WCET analysis. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2009.

19 Isaac Liu et al. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *ICCD*, September 2012.

20 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

21 Mingsong Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS)*, 2010.

22 André Oliveira Maroneze. *Certified Compilation and Worst-Case Execution Time Estimation*. PhD thesis, Université Rennes 1, June 2014.

23 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. IA3: An interference aware allocation algorithm for multicore hard real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

24 Adrian Prantl. *High-level compiler support for timing analysis*. PhD thesis, Technical University of Vienna, June 2010.

25 Jan Reineke et al. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2006.

26 Jan Reineke and Rathijit Sen. Sound and efficient WCET analysis in presence of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2009.

27 Marc Schlickling. *Timing Model Derivation – Static Analysis of Hardware Description Languages*. PhD thesis, Saarland University, January 2013.

28 Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for WCET analysis. In *Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2010.

29 Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1-2):265 – 286, 2008.

30 Martin Schoeberl et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, To appear in 2015.

31 Sanjit A. Seshia and Alexander Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Trans. Embed. Comput. Syst.*, 11(S2):55:1–55:27, August 2012.

32 Ingmar J. Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs*. PhD thesis, Saarland University, May 2010.

33 Theo Ungerer et al. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 99, 2010.

34 Theo Ungerer et al. parMERASA – multi-core execution of parallelised hard real-time applications supporting analysability. In *Euromicro Conference on Digital System Design (DSD)*, pages 363–370, 2013.

35 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

36 Petros Voudouris. Analysis and modeling of the timing bahavior of GPU architectures. Master's thesis, Eindhoven University of Technology, 2014.

37 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. 17:430–444, 2008.

38 Reinhard Wilhelm et al. 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.

39 Nicky Williams. WCET measurement using modified path testing. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2005.