Timing Analysis - From Predictions to Certificates

Nuno Gaspar[1], Simão Melo de Sousa[1,2], and Rogério Reis[2,3]

[1] RELEASE - Reliable And Secure Computation Group,
University of Beira Interior, Portugal,
{nmpgaspar, desousa}@di.ubi.pt,
[2] LIACC-UP
[3] DCC-FC, University of Porto, Portugal
rvr@ncc.up.pt

Abstract. In real-time systems, timing constraints must be satisfied in order to guarantee that deadlines will be met. The calculation of each task's worst-case execution time (WCET) is a prerequisite for the schedulability analysis, and hence of paramount importance for real-time systems. However, an accurate prediction can be difficult if the underlying hardware architecture possesses features like caches and pipelines.
In this paper we report our work in progress project on *ACCEPT*, an *Abstraction-Carrying CodE Platform for Timing validation*. Our approach counts on information gathered at source-code level (e.g. loop bounds, infeasible paths), defined by annotations that also express the intended timing behaviour. Furthermore, in the context of mobile code safety and in order to minimize the *trusted computing base*, we produce a checkable certificate whose validity entails compliance with the calculated WCET.

Keywords: real-time systems, worst-case execution time, abstract interpretation, abstraction-carrying code, fixpoint computation, timing validation

1 Introduction

Real-time systems can be seen as sets of tasks that are expected to perform some functionality under predefined timing constraints. In general, in order to ensure a correct system behaviour (schedulability analysis), an upper bound estimative for the worst-case execution time (WCET) of each task is necessary.

To improve its performance, modern processors include mechanisms like caches and pipelines which make instruction's execution time context dependent, increasing the difficulty of static timing analysis. To cope with this, one could be tempted to always assume the local worst case scenario (e.g. cache miss) in order to obtain safe predictions, but two problems could arise with such approach. On one hand, it could lead to an excessive over-approximation of the actual WCET, thus resulting in a waste of hardware resources. By other hand,

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 491–501

due to *timing anomalies* [RWT+06], the obtained prediction could in fact, be unsafe (an under-approximation).

While our platform considers the extraction of information from the source-code level (e.g. loop bounds, infeasible paths), in this paper we focus on the underlying mechanisms for the production of certificates and their validation process.

1.1 Motivation

The determination of safe and tight upper bounds for the WCET has been the object of intensive study in the literature [WEE+08]. Yet, there is no attempt, up to the present and to the best of our knowledge, to provide an independent validation mechanism w.r.t. the correctness of the predicted time bounds.

Mobile code safety has been progressively gaining notoriety in the sphere of real-time systems [SP01], both at research and industry levels, since they represent an enabling technology to tackle the limitations of standard client-server based approaches. In this context, in spite of the fact that previous methodologies are leveraged by the undertake of a formal approach, one would still have to put his faith on a potential *untrusted* third party, without being able to independently validate the correctness of the predicted time bounds.

One could argue that typically, applications loaded in embedded systems need not to satisfy real-time requirements (e.g. ring tones for mobile phones). However, in [KSH05], Kirsch et al identify the mobility of real-time programs as a challenging, but desirable feature for embedded systems. Indeed, even a software/system update can be seen as mobile code. Thus, being able to independently validate its timing behaviour would be of major interest for such systems. Moreover, a timing validation mechanism could also be a valuable asset for original equipment manufacturers and sub-contractors applications.

For instance, consider the following application scenario. There are several embedded systems that due to the functionality that they are expected to carry out, cannot be easily reachable. Coral sensors or control computers for satellites are examples of such embedded systems, where even a routine software/system update can be considered as mobile code. This updated software is also expected to satisfy stringent timing constraints, i.e. behave as mobile real-time code. Thus, being able to independently validate its timing behaviour would be of major interest for such systems.

This lack of an independent validation process is the issue that we address in this paper.

1.2 Related Work

There are numerous approaches in the literature focused in algorithms and tools for the derivation of the WCET [WEE+08]. Our goal here is not to present yet another approach of that type, but rather to emphasize how to produce a certificate that can be then used to validate the predicted time bounds.

Nevertheless, we should refer that the problem of certifying resource consumption, namely execution time, has already been addressed before, like in the work of Crary and Weirich [CW00], that use an extended type system capable of specifying and certifying bounds on resource consumption. However, this work makes no effort to determine bounds on execution times, but rather provides a mechanism to certify those bounds (for instance, obtained via a previous program analysis). The result of their approach is an executable that is certified w.r.t. resource consumption.

Furthermore, Bonenfant et al [BFHH07] present an interesting combination of information retrieved at source code level, with low-level timing information gathered with AbsInt's aiT tool [FH04]. This work provides guaranteed bounds on worst-case execution times for a strict, higher-order programming language.

The *Mobility, Ubiquity and Security* (MOBIUS) [BBC+06], and the *Mobile Resource Guarantees* (MRG) [Gua05] research projects also aim at the certification of resource consumption, their approaches rely mostly on theorem proving, whereas ours relies on abstract interpretation.

1.3 Contributions

The work reported here is included in a broader effort to provide a source level feedback on a BCET and WCET computation platform with certificate generation in the context of mobile code. This platform considers a high-level annotation language, preserving its semantics through a *WCET-aware* compilation process, and a *back-annotation* mechanism [HK07]. However, details on these features will be reported elsewhere.

In this paper, we focus on the low level interface. In this sense, this paper is a work in progress report on the BCET and WCET certificate generation and validation, and it intends to introduce and justify the underlying architecture.

1.4 Organization of the Paper

The remainder of this paper is organised as follows. Section 2 presents our architecture proposal towards an *Abstraction-Carrying Code* [APH04] platform. We explain the emergence of the certificate and how it can be used for independent validation of the calculated WCET in Section 3. Finally, conclusions and future work are discussed in Section 4.

2 Proposed Architecture

The general framework of our proposal is as illustrated by figure 1. We begin by extending the C programming language with annotations, following a *Design-by-Contract* [LC04] approach, that define the intended timing properties for each function. The timing specification of the main function is of most importance, however one may also want to define some constraints on the auxiliary functions. Moreover, by placing these annotations directly into the source code, we are also

able to express valuable informations for the subsequent WCET analysis, such as infeasible paths and loop bounds. This is achieved by the use of a WCET-aware compilation process, targeting the ARM instruction set, that preserves the annotations semantics.

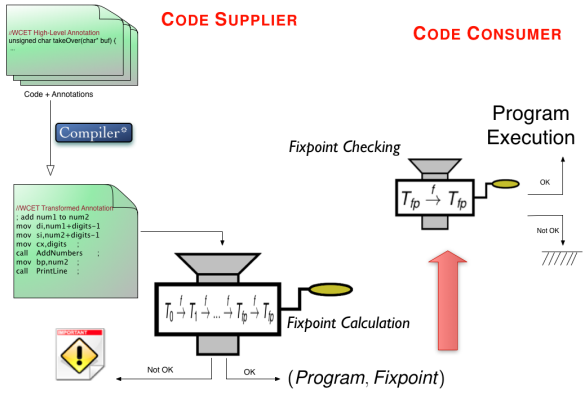

Fig. 1. Abstraction-Carrying Code based BCET/WCET Platform

Only at the hardware level one can accurately calculate the WCET, but feedback about the compliance of the given timing specification should be done at source-code level. Hence, in order to perform timing validation w.r.t. the functions' timing specification, we perform the WCET analysis at machine-code level, taking into account the effects of the hardware specific features, and alert of any possible non-compliance throughout the use of *back annotations* [HK07]. This is what is represented by the bottom left area of Figure 1. The idea of this mechanism is to propagate the timing information back to the source-code level, warning the system developers about the violation w.r.t. the timing specification. However, as stated in subsection 1.3, details on this mechanism will be reported elsewhere.

Let the set of execution times of a program $\mathcal{P}$ be denoted by $[\![\mathcal{P}]\!]$. The problem of verifying the compliance of the given timing specification can be thus formulated as follows:

$$\mathcal{P} \textit{ respects the timing specification } \mathcal{I} \textit{ if } [\![\mathcal{P}]\!] \subseteq \mathcal{I},$$

where $\mathcal{I}$ stands for the intended timing behaviour, i.e., the set of accepted execution times. The idea is to express the *collecting semantics* [WW08] of $\mathcal{P}$ as the fixpoint of a set of recursive equations. In general, however, the state space to be considered is too large to exhaustively explore all possible executions and some abstraction of the application domain is required in order to make the timing analysis feasible. With this in mind, our approach relies on abstract interpretation as the underlying technique. With its use, not only it provides us

an adequate framework to reason about, but also with an elegant way to infer an abstract model of the program that can play the role of a certificate.

2.1 Abstract Interpretation

In the abstract interpretation framework, a program $\mathcal{P}$ is interpreted over a simpler *abstract domain* $\mathcal{D}_\alpha$. This abstract domain permits to trade efficiency over precision, i.e., although it is an approximation, by computing the fixpoint over this abstract domain, we will be able to produce precise, yet safe, (over-)approximations of the collecting semantics.

The fixpoint calculation over this abstract domain will allow us to safely predict the processor behaviour for the program's execution (e.g. cache miss). With that information, and following the approach from the standard WCET architecture [Wil04], we can calculate the WCET, by determining its path through its control-flow graph. This is achieved by solving its corresponding *integer linear program* maximized for execution time. The process of determining the *best-case execution time* (BCET) is performed analogously.

Let $[\![\mathcal{P}]\!]_\alpha$ be the set of execution times calculated over the abstract domain $\mathcal{D}_\alpha$. It is clear that $[\![\mathcal{P}]\!]_\alpha$ is lower- and upper-bounded by BCET and WCET, respectively. Moreover, since the comparison between actual and intended semantics is easier if done in the same domain, we assume that the intended timing specification is also given in the abstract domain, i.e., $\mathcal{I}_\alpha \in \mathcal{D}_\alpha$.

The problem of verifying the compliance with the given timing specification can now be reformulated as

$$\mathcal{P} \textit{ respects the timing specification } \mathcal{I}_\alpha \textit{ if } [\![\mathcal{P}]\!]_\alpha \subseteq \mathcal{I}_\alpha.$$

At this stage, feedback regarding any possible non-compliance of $\mathcal{P}$ w.r.t. the timing specification can be reported through the use of back annotations [HK07], allowing the system developers to proceed accordingly.

2.2 Abstraction-Carrying Code

Proof-Carrying Code (PCC) [Nec97] is a general mechanism enabling a program consumer to locally check the validity of the code w.r.t. some safety policy. The inherent key benefit is that there is no need to trust any third party. However, there are three essential challenges for PCC to be used in practice:

(i) definition of *expressive safety and functionality policies*,
(ii) automatic generation of the certificate, i.e., proving the program correct, and
(iii) efficient certificate checking in the consumer side.

In the context of mobile code safety, most approaches rely on theorem proving, whereas Abstraction-Carrying Code (ACC) [APH04] relies on abstract interpretation.

In ACC, and in particular for the purposes of our platform, the above challenges are addressed by (i) getting hold of the effects that the processor specific

features (e.g. caches, pipeline) have on the execution time, which has already been addressed in the literature [The04,GR09]; (ii) using a fixpoint static analyzer to automatically infer an abstract model of the program, which can be then used as a certificate; and (iii) by a simple, easy-to-trust fixpoint checker.

3 Certificate Production and Validation

Let us now elaborate on this process applied to our platform proposal. A program is characterized by its control-flow graph, constituted by a set of edges $E \subseteq V \times Ins \times V$, where V represents the program points, $v_i \in V$ models the program's entry point and Ins models the instruction to be executed whenever taking that edge.

A semantic function $[\![.]\!] : Ins \to (S \to S)$ assigns to each $ins \in Ins$, a transfer function that models its effect on the program state S, being evaluated. For instance, in the ARM instruction set, the instruction B *address* is specified as $R15 := address$, i.e., update of the program counter register to the address given by the evaluation of the expression *address*, and thus would have to be modelled accordingly to its specification.

The collecting semantics assigns for each program point V, the set of program states S, which may occur in any possible execution, i.e., $CS : V \to \mathcal{P}(S)$ (where $\mathcal{P}(S)$ stands for powerset of S). The analysis to be performed can be specified by extracting a number of equations from the program being considered. There are two types of equations. The first one, relates exit with entry information for each program point V. While the second, relates entry information of a program point V_i, with exit information of nodes from which there exists an edge to the program point V_i, i.e., $\bigcup\{V_j \mid (V_j, ins, V_i) \in E\}$.

The resulting system of equations can be solved by computing the least fixpoint $lfp(F) = F^n(\lambda v.\emptyset)$ of the functional $F : (V \to \mathcal{P}(S)) \to (V \to \mathcal{P}(S))$:

$$F(f)(v') = \begin{cases} S_0 & \text{if } v' = v_{in}, \\ \bigcup_{(v,ins,v') \in E} [\![ins]\!](f(v)) & \text{otherwise,} \end{cases} \tag{1}$$

where $S_0 \subseteq S$ is the set of the program's initial states.

However, as mentioned in Section 2, computing the collecting semantics of a large and complex program $\mathcal{P}$ can be too much expensive to be feasible. Hence, the analysis is performed on a simpler abstract domain $\mathcal{D}_\alpha = (S, L, \beta, \gamma)$, where $L = (L, \sqsubseteq, \bigsqcup, \bot, \top)$ is a complete semi-lattice and $\beta : S \to L$ is a *representation function*, mapping concrete to abstract states. The idea, is that β maps a state S to the best property describing it. Finally, $\gamma : L \to \mathcal{P}(S)$ is a *concretization function* mapping abstract states to concrete states.

As we have seen above, the collecting semantics operates over sets of states, while our abstract domain, operates over sets of properties. Thus, with the purpose of relating these two domains, we define an *abstraction function* $\alpha : \mathcal{P}(S) \to L$, by $\alpha(S') = \bigsqcup\{\beta(s) \mid s \in S'\}$. The concretization function γ, and the abstraction function α, will therefore yield the following relation:

$$\mathcal{P}(S) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} L$$

The above relation is defined such that $\alpha(X) \sqsubseteq l \Leftrightarrow X \subseteq \gamma(l)$, and thus establishing the pair (α, γ) as a *Galois connection*. Furthermore, in order to ensure termination we require the *Ascending Chain Condition* to hold, i.e., every ascending chain of elements eventually terminates. For this, both the abstraction function α, and the concretization function γ, must be monotonic w.r.t. the $\sqsubseteq$ and $\subseteq$ operators, respectively.

The semantic function defined above, can now be redefined as an *abstract semantic* function $[\![.]\!]_\alpha : Ins \to (L \to L)$, over the abstract domain. The abstract counterparts of the transfer functions $[\![ins]\!]$, i.e., $[\![ins]\!]_\alpha$, must also be monotonic w.r.t. the $\sqsubseteq$ operator. Finally, the analysis can now be applied with the *abstract collecting semantics* $CS_\alpha : V \to L$, such that $\forall v \in V : CS(s) \subseteq \gamma(CS_\alpha(v))$, i.e., the computed results are either precise or an over-approximation of the collecting semantics, and thus are safe.

The resulting system of equations can be solved by computing the fixpoint $lfp(F_\alpha) = F_\alpha^n(\lambda v.\bot)$ of the functional $F_\alpha : (V \to L) \to (V \to L)$:

$$F_\alpha(f)(v') = \begin{cases} l_0 & \text{if } v' = v_{in}, \\ \bigsqcup_{(v,ins,v') \,\in\, E} [\![ins]\!]_\alpha(f(v)) & \text{otherwise,} \end{cases} \tag{2}$$

where the abstraction of the *concrete* initial states is defined as initial abstract state, thus $\alpha(S_0) \sqsubseteq l_0$.

It should be clear that, since the abstract transfer functions, $[\![ins]\!]_\alpha$, are monotonic w.r.t. the $\sqsubseteq$ operator, by induction we obtain $F_\alpha^n(\lambda v.\bot) \sqsubseteq F_\alpha^{n+1}(\lambda v.\bot)$ for all n. All the elements of the sequence are in L, and since this is a finite set, not all elements of the sequence can be distinct. Thus, there must be some n such that:

$$F_\alpha^{n+1}(\lambda v.\bot) = F_\alpha^n(\lambda v.\bot)$$

Furthermore, since $F_\alpha^{n+1}(\lambda v.\bot) = F_\alpha(F_\alpha^n(\lambda v.\bot))$, we have reached the least fixpoint of F_α, i.e., $lfp(F_\alpha)$, and thus found a solution to the equation system.

The analysis to be instantiated depends on the target processor being evaluated. In our current prototype implementation of *ACCEPT*, we focus ourselves in the ARM7TDMI-S and ARM920T processors. While for the former only a pipeline analysis is performed that captures the instruction's overlapping effect, for the latter, since it also features a cache memory, an integrated cache and pipeline analysis is performed [The04].

3.1 Program Producer - Certificate Production

After computing this fixpoint, and thus having the cycles counts for each basic block of the control-flow graph, we are able to calculate both the BCET and

WCET by means of integer linear programming techniques, and thus verify the compliance with the giving timing specification (Figure 1). However, we can also let the obtained fixpoint play the role of a certificate.

In the context of mobile code safety one cannot trust the origin of the program. Hence, by adding to the code the certificate and sending both to the program consumer, it can be performed a local and independent check of the program's timing behaviour, thereby avoiding the need to trust in the code producer.

3.2 Program Consumer - Certificate Validation

The program consumer receives a program along with its certificate. In order to check the compliance with the intended timing specification, the first step is to compute the program's control-flow graph and verify that the certificate is a valid *abstraction*. Then, since the certificate is supposedly a fixpoint, the checking procedure can be written as:

$$Check(Certificate) = \begin{cases} True & \text{if } F_\alpha(Certificate) = Certificate, \\ False & \text{otherwise.} \end{cases} \quad (3)$$

Since the certificate is supposed to be a fixpoint, another iteration over it cannot change anything, thus, on the program consumer side, a simple one-pass computation is suficient to check that the certificate is indeed a fixpoint.

In the cases where the received certificate does not behave as a fixpoint, the program consumer can simply reject the program. One could argue that we could let the program run, and *kill* its execution in the case of a timing behaviour non-compliance. However, that would be a waste of resources, and in the context of embedded systems, which tend to have very limited computational resources, it is unacceptable. On the other hand, if the certificate is indeed a fixpoint, then the program consumer can locally compute the BCET and WCET by standard integer linear programming techniques, and thus check the compliance with the timing specification. Furthermore, it should be noted that in this framework, it is also possible for the program consumer to define new timing policies. For instance, one can be interested in tightening the timing constraints.

This validation process requires that both the producer and consumer share the same abstract transfer functions. Indeed, if the consumer used different abstract transfer functions the certificate checking process would be inefficient, and thus prohibitive for such scarce resource equipments as embedded systems. One could argue that the independence in the timing validation process is compromised by that fact, however, it should be noted that the trusted computing base is limited to this checking operation, i.e., a simple, easy-to-trust fixpoint checker, that only has to perform a one iteration process. Hence, this approach allows to detect if a program has been tampered with, since an adulteration in the program code would be detected when performing the checking operation, i.e.,

the fixpoint iteration. In the context of mobile code, this is particularly relevant since, rather than simply put a blind confidence on a previous timing analysis, one can validate the program's timing behaviour by solely relying on a fixpoint checker.

4 Conclusions and Future Work

Abstract Interpretation has been widely used in the industry, being static timing analysis one of its most successful applications [WW08]. In our approach we also use the Abstract Interpretation framework as the underlying technique, we obtain our BCET and WCET predictions taking into account the hardware specificities (cache, pipeline) [The04,GR09], by explicitly following a standard fixpoint computation strategy [Kil73], and then apply standard integer linear programming techniques in order to compute the BCET and WCET. This fixpoint computation will allow us to infer an abstract model of the program, which can then be used as a certificate, i.e., a program consumer can locally validate the received program w.r.t. to its timing behaviour, by simply checking that this abstract model is indeed a fixpoint (a one-pass process), and then compute the BCET and WCET with the received certificate.

This paper is a work in progress report on the timing certificate generation and validation and intend to introduce and justify the underlying architecture. We presented our architecture proposal for *ACCEPT*, an *Abstraction-Carrying CodE Platform for Timing validation*. In our prototype being implemented, we avoid a *binary-to-assembly* translation phase, by making our compilation process directly produce ARM assembly.

At this stage there are still some open issues that remain to be addressed. One of the main challenges that we face in order to make our *ACCEPT* platform useful in practice is the size of the produced certificates. Embedded systems are known for their scarce resources, and thus, cannot afford to waste computational means. In [AASPH06], Albert et al introduce the notion of a *reduced certificate*, with the objective of producing a certificate that only contains the essential information which the program consumer cannot reproduce by itself, while not yielding an overhead in the certificate checking process.

Our actual focus on the certification generation part of the *ACCEPT* platform is now on the pragmatical evaluation of our proposal. For now we are not concerned with performance, but with correctness and adequacy (in the context of mobile code). However, a BCET/WCET platform is only useful if it provides tight and safe time bounds. In this sense, we use state of the art algorithms for their calculation. Nevertheless, comparing equitably this kind of platform against reference tools [FH04], even pragmatically in the form of benchmark, is not a trivial task, the same source-code, compilation process and/or low-level code and target architecture must be considered. However, we plan to report on case studies and practical results of our framework very soon.

To the best of our knowledge this is the first work applying the concepts of Abstraction-Carrying Code to the static timing analysis field.

Acknowledgments

This work is partially supported by the RESCUE project PTDC/EIA/65862/2006 funded by FCT (Fundação para a Ciência e a Tecnologia). We also thank Vitor Rodrigues for his valuable suggestions and comments.

References

[AASPH06] Elvira Albert, Puri Arenas-Sánchez, Germán Puebla, and Manuel V. Hermenegildo. Reduced certificates for abstraction-carrying code. In *ICLP*, pages 163–178, 2006.

[APH04] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. Abstraction-carrying code. In *LPAR*, pages 380–397, 2004.

[BBC+06] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. Mobius: Mobility, ubiquity, security. In *TGC*, 2006.

[BFHH07] Armelle Bonenfant, Christian Ferdin, Kevin Hammond, and Reinhold Heckmann. Worst-case execution times for a purely functional language. In *In 18th IFL 2006*. Springer, 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252. ACM, 1977.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *POPL '00*, pages 184–198, New York, NY, USA, 2000. ACM.

[FH04] Christian Ferdinand and Reinhold Heckmann. ait: worst case execution time prediction by static program analysis. In *IFIP Congress Topical Sessions*, pages 377–384, 2004.

[GR09] Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In *SAS'09*, pages 120–136. Springer-Verlag, 2009.

[Gua05] Mobile Resource Guarantees. http://groups.inf.ed.ac.uk/mrg/, 2005.

[HK07] Trevor Harmon and Raymond Klefstad. Interactive back-annotation of worst-case execution time analysis for java microprocessors. In *13th IEEE RTCSA*, Washington, 2007. IEEE Computer Society.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73*, pages 194–206, New York, NY, USA, 1973. ACM.

[KSH05] Christoph M. Kirsch, Marco A. A. Sanvido, and Thomas A. Henzinger. A programmable microkernel for real-time systems. In *VEE '05*, 2005.

[LC04] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2004.

[Nec97] George C. Necula. Proof-carrying code. In *POPL '97*, pages 106–119, New York, NY, USA, 1997. ACM.

[RWT+06] Jan Reineke, Bjrn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Intl Workshop on WCET Analysis*, 2006.

[SP01] Alexander D. Stoyen and Plamen V. Petrov. Towards a mobile code management environment for complex,real-time, distributed systems. *Real-Time Syst.*, 21(1/2):165–189, 2001.

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

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[Wil04] Reinhard Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In *VMCAI'04, LNCS 2937*, 2004.

[WW08] Reinhard Wilhelm and Björn Wachter. Abstract interpretation with applications to timing validation. In *CAV '08*, 2008.