Automatic Identification of Timing Anomalies for Cycle-Accurate Worst-Case Execution Time Analysis

Jochen Eisinger and Ilia Polian and Bernd Becker
Albert-Ludwigs-University
Georges-Köhler-Allee 51
79110 Freiburg, Germany
{eisinger|polian|becker}@informatik.uni-freiburg.de
Alexander Metzner
Carl von Ossietzky University
26129 Oldenburg, Germany
metzner@informatik.uni-oldenburg.de
Stephan Thesing and Reinhard Wilhelm
Saarland University
66041 Saarbrücken, Germany
{thesing|wilhelm}@cs.uni-sb.de

Abstract— **Hard real-time systems need methods to determine upper bounds for their execution times, usually called worst-case execution times. Timing anomalies are counter-intuitive conditions in which a local speed-up of an instruction results in a global slow-down. Modern efficient timing analysis tools may yield inaccurate results when applied to processors with timing anomalies while methods which are suited for timing-anomalous systems are computationally expensive. Timing anomaly identification is key in choosing the right analysis technique for a given processor. In this paper, for the first time, an automated timing anomaly identification approach based on formal methods is presented. We validate the method by applying it to a simplified microprocessor using a commercial model checking tool.**

Keywords: Real-time system verification, Worst-case execution time (WCET), Timing anomalies, Formal methods

I. Introduction

Worst-case execution time (WCET) analysis is a major technique in verification of real-time systems, i.e. ensuring that the system will perform a given task not exceeding a given time limit [1]. WCET of a system's tasks is also required for schedulability analysis of the system. There are several WCET concepts. Under the *unit time abstraction* every assembler instruction is assigned the duration of one and consequently WCET analysis is reduced to counting the number of executed instructions [2]. WCET has also been defined for higher abstraction levels when architecture-specific information is not employed [3]. In this work, we focus on a more accurate WCET definition: *cycle-accurate WCET*, which takes into account that assembler instructions may require different number of cycles to be executed. Moreover, the number of cycles may depend on the operands and/or the system state (e.g., whether or not an operand is in cache). Due to these non-trivial dependencies, cycle-accurate WCET cannot be computed exactly in general, so WCET estimates are used. While WCET underestimations are unacceptable, overestimations cannot be avoided in general. There are significant efforts to derive overestimations that are as tight as possible [4].

For some processor architectures with variable instruction execution times, the worst-case execution time of an individual instruction may not contribute to the longest possible execution time of the program. Similarly, a local short execution time may lead to a worse global time. Such behavior is referred to as a *timing anomaly* [5]. An instance of a timing anomaly is a cache miss which results in a shorter global execution time than a cache hit. Some WCET estimation methods consider the control-flow graph, assign worst-case local execution times to the nodes of the graph and try to construct a path through the graph such that the sum of the local WCETs on the path is maximal. The global WCET is given by that sum. While such methods are computationally efficient, they are not applicable to systems that are prone to timing anomalies. Hence, it is important to know whether timing anomalies can occur in a system before performing WCET analysis. If timing anomalies cannot be ruled out, computationally expensive approaches such as exhaustive enumeration of all possible paths may be required. Alternatively, techniques such as program code modification [5] may be employed to make efficient and tight WCET estimation possible.

Many classes of microprocessors cannot have timing anomalies because they lack architectural features necessary to induce these non-trivial conditions. For all other processors, the susceptibility to timing anomalies is presently determined in a long and error-prone process. In this paper, we present for the first time an automatic method that identifies timing anomalies in a given processor using *model checking*. For the analyzed processor (represented in VHDL), a *property* is derived such that input sequences satisfying this property correspond to a program which exposes a timing anomaly. The input sequences are determined using model checking. Model checking has recently been suggested for use in WCET analysis itself [6]. However, its application to timing anomaly identification is new.

As a case study, we applied our method to a simple microprocessor with a Tomasulo scheduler, which is prone to timing anomalies due to out-of-order instruction scheduling. We present the property for timing anomaly identification for this processor and report the application of commercial bounded model checking software.

The remainder of the paper is organized as follows. The next two section reviews the basics of timing anomalies. Section III presents the proposed method. Experimental results on a case study are reported in Section IV. Section V concludes the paper.

II. Timing Anomalies

An instruction may require a different number of cycles to execute depending on the current state of a processor, e.g. whether or not an operand is in the cache. Timing anomalies are counter-intuitive influences of the (local) execution time of a single instruction on the (global) overall execution time of the whole program. If the execution takes ΔT cycles less in one processor state than in an other, a timing anomaly is given if either the overall program runs longer in the first processor state, or if it is accelerated by *more* than ΔT cycles. Of particular concern are cases in which a local acceleration may result in a global slowdown. In order to illustrate this phenomenon, an example from [5] is briefly repeated here.

A sequence of five instructions A; B; C; D; E is run on a simplified PowerPC architecture. A is a load instruction which requires 2 cycles if the load address hits the cache and 8 cycles otherwise; B and C are ADD instructions which take one cycle each to execute; and D and E are MUL instructions executed by the multiple-cycle integer unit with latency of 4 cycles. B depends on the result of A, D depends on the result of C, and E depends on the result of D. Out-of-order execution is possible for ADD and MUL instructions.

Figure 1 shows the activity of the three units needed for executing the program if instruction A hits the cache (above) and if A misses the cache (below). In the former case, B is executed immediately after A (in cycle 3). B is preferred over C because it is older. Consequently, C can start only in cycle 4 and D and E have to wait for the result of C because of dependencies. In the latter case (cache miss) B must wait for the result of A (cycle 11). Hence, C is scheduled in the cycle in which it is issued (cycle 3), and D and E are executed one cycle earlier. The overall execution time is reduced from 12 to 11 cycles despite the cache miss.

Timing anomalies are not an issue if the complete state of the processor and all its inputs are known, as the processor is a deterministic system and there is only one possible execution path with well-defined execution time. As a consequence, determining the execution time for every possible state and input sequence and taking the maximum value would result in an accurate WCET. Clearly it is not an option for any realistic processor as the state and input space are huge. Hence, the WCET estimation methods use approximate information. For instance, in the example above there is no exact knowledge on the cache content, so it is impossible to tell whether the load operation will result in a cache hit or miss and thus how many cycles it will take. An efficient analysis method would assume the worst-case behavior, which is incorrect in the example. Due to the timing anomaly, the correct WCET estimation method needs to consider both cache hit and miss, determine WCET for both cases and return the larger value as the global WCET. Clearly, if there are many such ‘forks’, the method has to consider a high number of paths potentially leading to global WCET. Every situation for which timing anomalies are not ruled out and in which n different option exist will increase the number of paths to be considered by the factor of n. That is why the identification of timing anomalies, in particular proving their absence, can significantly reduce the complexity of the WCET algorithm. If timing anomalies can be ruled out altogether, only one path needs to be considered and local worst case can be safely assumed on every node along this path.

Not all possible timing anomalies will invalidate WCET analysis. We call timing anomalies which may have implications on WCET estimation *strong timing anomalies* and timing anomalies which will never invalidate WCET results *weak timing anomalies*. An instance of a weak timing anomaly is a local speed-up resulting in an even larger global speed-up [5]. This timing anomaly is weak because it does not invalidate WCET results if worst-case local behavior is assumed. Since the purpose of this work is to improve the WCET methodology, we focus on strong timing anomalies (which will be introduced more formally in Section III).

For WCET analysis of a processor, it is important to

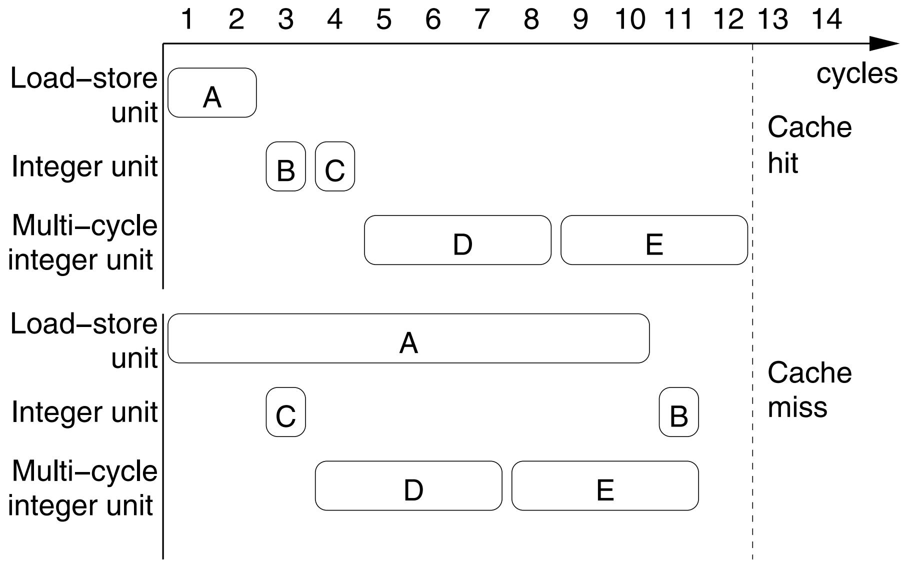

Fig. 1. Timing anomaly example from [5]

know whether it can have timing anomalies. State-of-the-art WCET techniques perform microarchitecture analysis first and then determine the worst-case path. If the microarchitecture analysis relies on the premise that local worst-case scenarios lead to global worst-case execution times, it cannot be used for processors with timing anomalies. On the other hand, considering all possible local scenarios for microarchitecture analysis is conservative but leads to a much more expensive timing analysis.

Note that the number of extra cycles due to a timing anomaly may not be bounded (domino effect). An example for this effect has been shown in [5], and Schneider [7] demonstrated it for the Motorola PowerPC 755.

III. Automatic Timing Anomaly Identification

In the existing literature, timing anomalies are defined in an semi-formal way [5]. To handle timing anomalies by model checking, a mathematical formalization is required. As mentioned above, we are interested only in strong timing anomalies, i.e. ones which may have implications for WCET analysis. A strong timing anomaly is present if a program run where local worst-case is assumed for every instruction is faster than some other run of the same program. Consequently, we focus on instances where a local speed-up results in a global deceleration. Although other types of timing anomalies (weak timing anomalies) are known, they are not a primary concern for WCET analysis.

We distinguish between two problems: *code-specific timing anomaly identification* (for a given microprocessor and a given piece of program code (software) to be executed on that processor), and *code-independent timing anomaly identification* (determining whether the microprocessor as such is prone to timing-anomalous behavior). In the code-specific case, the existence of two different execution paths through the code which expose a timing anomaly is investigated. For the code-independent case, we define a processor to be *timing-anomalous* iff there is *at least one* code sequence for which a timing anomaly occurs. The code-independent problem is more challenging than the code-specific problem because it has a larger solution space. On the other hand, it is also more general: if the absence of timing anomalies has been proven for a microprocessor, this means that this microprocessor's features are incapable of creating a timing anomaly no matter which software is run on the processor. As a consequence, efficient WCET tools which assume the absence of timing anomalies can safely be applied for any code on this processor. If a processor has been found to be timing-anomalous, it may be worth running code-specific timing anomaly identification for a given software, because the timing anomaly may not occur for this code (in that event, efficient WCET determination is available).

In this paper, we target the more difficult problem of code-independent timing anomaly identification which produce results valid *for arbitrary software*. Our method is constructive, i.e., we determine a piece of code which exposes the timing-anomalous behavior as a part of the solution (along with a detailed trace). Note that other publications [5] consider the code-specific version of the problem (and their approach is not based on formal methods). In our approach, it is possible to add constraints which specify the instruction sequence. By doing so, it is possible to solve also code-specific problem instances using the same framework.

For identifying strong timing anomalies in a microprocessor, the processor is modeled as a finite state machine.

Two almost identical instances of the processor, CPU_f and CPU_d, are instantiated. The difference between CPU_f and CPU_d is that all features that might cause a timing anomaly are disabled for CPU_d (f stands for "full-feature" and d stands for "disabled"). For instance, CPU_d may not access the cache, i.e. every memory access results in a cache miss for CPU_d (but not necessarily CPU_f). Our goal is to find an input sequence (assembly program) which will lead to a strong timing anomaly (i.e. there is a cycle t_0 in which CPU_f overtakes CPU_d and a later cycle t_1 in which CPU_d overtakes CPU_f), or to prove that no such input sequence exists. This is accomplished by formulating a property (described below) and running a model checking tool for this property.

We require that CPU_f and CPU_d have an *instruction counter* IC. We refer to the instruction counter of CPU_f (CPU_d) as IC_f (IC_d). For a cycle t, $IC_i(t)$ gives the number of instructions that have been completed by CPU_i up to this cycle ($i \in \{f, d\}$). Note that IC does not correspond to the program counter PC which contains the memory address of the next instruction to be read by the CPU.

$IC_f(t) > IC_d(t)$ means that in cycle t, CPU_f executed more instructions as CPU_d, i.e., CPU_f has *overtaken* CPU_d. If there is no register IC in the processor itself, it must be added. Note that it is required only for analysis (identification of timing anomalies), so it is not actually implemented in hardware. The needed functionality is very simple (a few lines of VHDL code), and its insertion can be automated. After the timing anomaly identification procedure is complete, the added register IC can be removed from the design model again.

Now the property for timing anomaly identification is described. We assume that the input to the processor is a stream of instructions. $Input_i(c)$ ($c \in \mathbb{N}$) denotes the c^{th} instruction received by CPU_i (note that the instructions may be executed out-of-order, so the c^{th} received instruction is not necessarily the c^{th} executed instruction). $State_i(t_0)$ denotes the state of CPU_i in cycle t_0. However, the property considers only the *relevant part of the system state*, which does not include register file, cache and branch predictors. This is formalized by having a projection π_{Rel} on the relevant part of the state. $\pi_{Rel}(State_i(t_0))$ is the relevant part of the state of CPU_i in cycle t_0. The property is:

$$\begin{aligned}\forall t_0 \in \mathbb{N} : \quad & \Big[\pi_{Rel}(State_f(t_0)) = \pi_{Rel}(State_d(t_0)) \\ & \wedge \big(\forall c \in \mathbb{N} : Input_f(c) = Input_d(c)\big) \\ & \wedge IC_f(t_0) > IC_d(t_0)\Big] \\ & \Longrightarrow \forall t_1 \geq t_0 : IC_f(t_1) \geq IC_d(t_1). \qquad (1)\end{aligned}$$

The property states that CPU_f has no strong timing anomaly (if a strong timing anomaly exists, model checking will yield a counterexample). In more detail, if CPU_f and CPU_d start in the same states and on the same instruction stream and CPU_f has overtaken CPU_d in cycle t_0 (local speed-up), CPU_d will never overtake CPU_f (no global slow-down). Note that CPU_d is *not* the actual processor under consideration but an artificial finite state machine used to provide a reference point. The modifications done in CPU_d correspond to the assumptions which the WCET algorithm makes about the CPU under consideration (CPU_f). Since the WCET algorithm does not take register or cache contents into account, it would never consider the speed-up of the MUL instructions but rather assume local worst case.

CPU_d cannot have a timing anomaly because it does not have varying execution times for instructions (but the processor under consideration is definitely not restricted to static instruction times). The register files, caches and branch predictors of CPU_f and CPU_d may differ as pointed out above. The functional outputs of the processors are ignored by the property; only the instruction counters IC_f and IC_d are used.

If the property from Eq. (1) has been falsified by model checking, the counterexample yields an input sequence (assembly program) which causes a timing anomaly due to an enabled feature (i.e. CPU_d with the feature disabled is behind CPU_f in cycle t_0 but overtakes it in a later cycle t_1). Note that a local speed-up on CPU_d is impossible, so it is sufficient to consider timing-anomalous behavior of CPU_f.

If the property has not been falsified, it is a formal proof that the strong timing anomaly will not occur for any input sequence. This means that the WCET tool can safely assume that the CPU behaves like the hypothetical processor CPU_d. This simplifies the WCET analysis because local worst case can be assumed for every instruction. Since there are provably no timing anomalies, the resulting WCET is a valid upper bound, i.e., it cannot fall below the accurate execution time for any input sequence. As a result, the WCET tool run time is reduced without sacrificing accuracy.

The approach presented so far handles the code-independent timing anomaly identification. If the program code is known, the property from Eq. (1) is easily modified to incorporate that information: Let the program sequence be $Instr_1; Instr_2; \ldots, Instr_m$. Instead of requiring $Input_f(c) = Input_d(c)$, the modified formula is then $Input_f(c) = Input_d(c) = Instr_c$. Note that the solution space is reduced by this restriction, which is advantageous for efficiency.

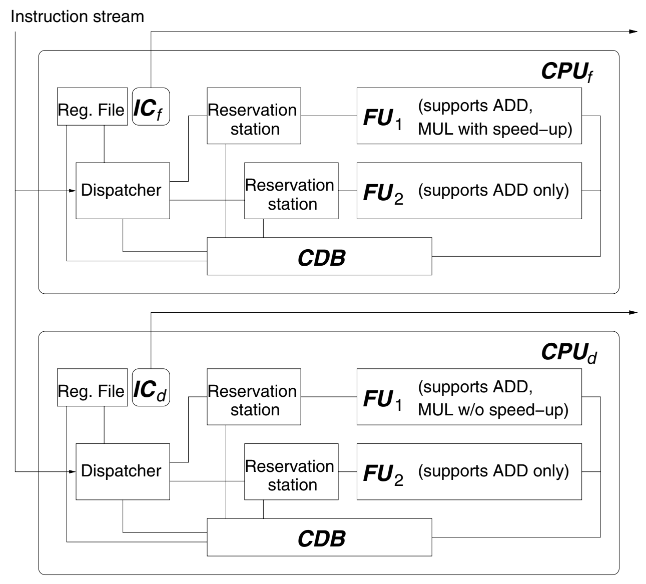

Fig. 2. Two instances of the example processor for property checking

IV. Experimental Results

In this section, we consider a processor with two functional units (FU_1 and FU_2) and a Tomasulo-style scheduler [8]. FU_1 can execute ADD and MUL instructions, and FU_2 can only execute ADD instructions. The execution of an ADD instruction always takes 6 cycles; the execution of a MUL instruction takes 14 cycles. As a speed-up, the execution of a MUL takes only 5 cycles if one of the arguments is 0 (a similar speed-up also exists in certain PowerPC architectures). The result of either FU is written on a Common Data Bus (CDB). This takes 2 cycles if the CDB is available immediately (i.e. not being written by the other FU). The values on the CDB are written back into the register file which consists of eight data registers R0 through R7. If a value of a register that is on CDB but not yet written back is requested, it is taken directly from CDB.

After an instruction has been received, fetching, decoding and scheduling of an instruction takes 2 cycles until the execution can start if all operands are available. Consequently, if two instructions are scheduled on the same FU, the minimal time between these instructions is 4 cycles.

In order to determine whether the speed-up of the MUL instruction from 14 to 5 cycles can lead to a strong timing anomaly, we are employing bounded model checking (BMC) [9] as the model checking engine. BMC considers only input sequences of bounded length (where the bound is given by the *BMC constant* k), which makes it a semi-complete method. BMC is complete if the system under consideration has a *diameter* d, i.e. the maximal length of a relevant path with respect to the regarded property, which is less than k. The derivation of useful diameter estimates is a focus of our future research; in this paper we set k driven by the problem instance. Note that we use BMC for the sake of efficiency; it would be possible to use unbounded model checking tools which would result in a complete procedure.

We instantiate two copies of the processor, CPU_f and CPU_d. Both CPU_f and CPU_d have FU_1 and FU_2, but FU_1 of CPU_f supports the MUL speed-up and FU_1 of CPU_d does not support it. All other features of CPU_f and CPU_d are identical and the instruction counters IC_f and IC_d are provided (see Fig. 2). Then, the property from Eq. (1) is formulated and BMC (with $k = 64$) is run.

We used the tool CVE **gateprop** by OneSpin Solutions for BMC. Its input language supports keywords such as `during` (to specify that a condition holds within a time interval), which makes it very convenient to formulate Eq. (1).

It turns out that BMC can find a counterexample. The automatically generated assembly program is given below (the destination register is on the left hand side):

```
1:  ADD R7, R0, R1
2:  MUL R6, R4, R6
3:  ADD R7, R0, R1
4:  MUL R4, R7, R7
5:  ADD R6, R7, R7
```

The output of the property checker includes a full trace demonstrating the timing anomaly. Figure 3 shows parts of the trace, namely the instructions processed by FU_1 and FU_2 of CPU_f and CPU_d and the instruction counters IC_f and IC_d. Recall that FU_1 can execute ADD and MUL instructions while FU_2 can execute only ADD instructions, and that a speed-up can be employed by FU_1 of CPU_f if an operand of MUL is 0 but this speed-up has been disabled for CPU_d. Also recall that IC_f and IC_d contain the number of instructions executed so far.

It is clearly seen that initially CPU_f overtakes CPU_d, e.g. in cycle $t_0 = 20$ ($IC_f = 2$, $IC_d = 1$) because CPU_f can utilize the speed-up for executing instruction 2 which CPU_d cannot. However, in cycle $t_1 = 41$ CPU_d overtakes CPU_f ($IC_f = 3$, $IC_d = 4$) although it cannot use the speed-up. Notice that the first 11 cycles are consumed by setup activities aimed at bringing the processor into a reachable state. Shaded semicircles indicate the instruction decoding and the write-back. Both require two cycles, with exception of instruction 1 in CPU_f, which cannot be written back in 2 cycles because the CDB is occupied by instruction 2 and FU_1 has a higher priority. The computed result can be written on the CDB only in cycle 18. The same situation occurs for instruction 3 in CPU_d.

By using BMC, a strong timing anomaly could be identified for the example processor. A trace clearly showing

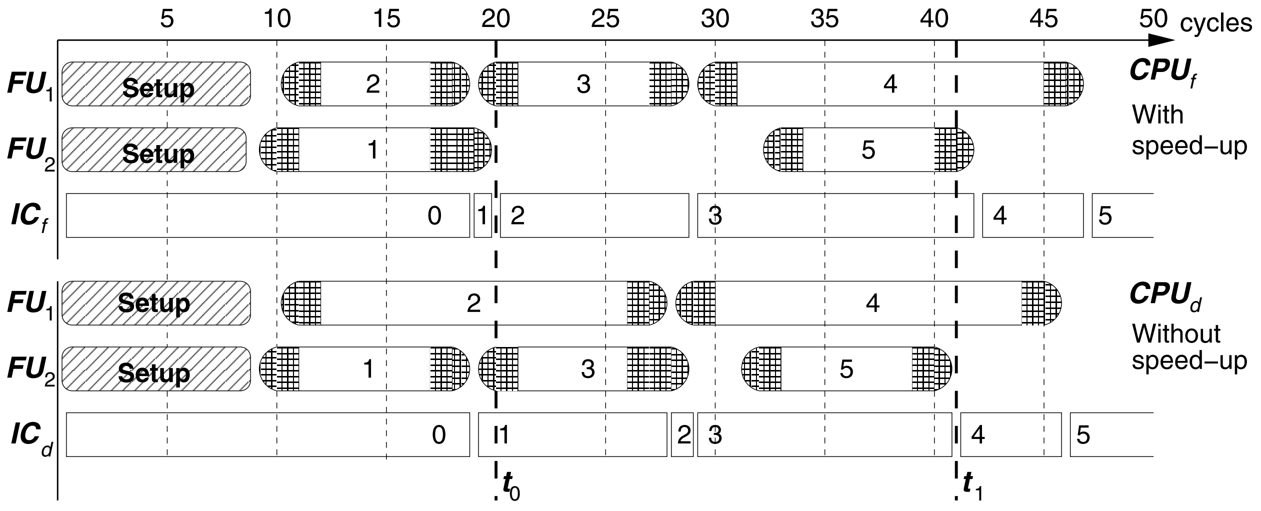

Fig. 3. Trace generated by BMC demonstrates a strong timing anomaly

the anomalous behavior has also been generated. For such a processor, efficient WCET estimation techniques based on local analysis followed by worst-case path search could produce wrong results and should be avoided.

V. Conclusions

Timing anomalies may invalidate the results of efficient worst-case execution time analysis tools. As a consequence, it is important to know whether a given processor can exhibit timing-anomalous behavior. Currently, this is an error-prone manual task for which no tool support exists.

We presented, for the first time, an automatic method for timing anomaly identification. It is based on model checking, a highly successful technique from the field of formal verification. We presented the property to detect the timing anomaly. We reported the application of our method to a simplified timing-anomalous processor using a commercial bounded model checker. This yielded a detailed trace demonstrating the anomaly and hence the inapplicability of standard timing analysis tools. For a processor shown to be timing-anomalous, either more elaborate WCET analysis techniques must be used, or the feature that has led to the anomaly must be disabled.

We are currently planning to apply our technique to larger microprocessors. This will necessitate the use of abstraction techniques such as slicing which must be incorporated into the property and may lead to a completely new definition of a timing anomaly. A further point of investigation is the application of complete methods such as unbounded model checking or making BMC complete by a tight approximation of the diameter of the system.

Acknowledgments

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information. We are thankful to Prof. Dr. W. Büttner of OneSpin Solutions for making the CVE **gateprop** license available.

VI. References

[1] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[2] T. Schuele and K. Schneider. Exact runtime analysis using automata-based symbolic simulation. In *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 153–162, 2003.

[3] G. Logothetis and K. Schneider. Exact high level wcet analysis of synchronous programs by symbolic state space exploration. In *Design Automation and Test in europe*, pages 196–203, 2003.

[4] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.

[5] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time-Systems Symp.*, 1999.

[6] A. Metzner. Why model checking can improve WCET analysis. In *Int'l Conf. on Computer-Aided Verification*, volume 3114 of *LNCS*, 2004.

[7] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

[8] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. and Develop.*, 11(1):25–33, 1967.

[9] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.