WCET Overapproximation for Software in the Context of a Cyber-Physical System

Niklas Krafczyk* Heinz Riener* Goerschwin Fey*†
*German Aerospace Center
Germany, Bremen
{firstname.lastname@dlr.de}
†University of Bremen
Germany, Bremen
fey@informatik.uni-bremen.de

***Abstract*—We propose an approach for overapproximating the *worst-case execution time* (WCET) of embedded control software using formal methods. Model checking is iteratively applied to compute the WCET from the machine code of the software considering a platform and an environment model. We implemented the approach and present first experiments for a thermal controller application executed on a LEON3 processor under different environment constraints.**

I. Introduction

The concept of a *cyber-physical system* (CPS) has recently emerged as a model that tightly integrates digital computation with its physical environment. A discrete system continuously interacts with its physical environment via senors and actuators such that CPS describe closed-feedback loop systems.

From the application perspective, this reflects the view of traditional *feedback* or *closed-loop control systems*. For instance, consider a software controller for a real-time application executed on a hardware platform that interacts with its physical environment via sensors and actuators. The controller has to react on certain input values within a specified amount of time or otherwise safe operation of the overall system cannot be guaranteed. Thus, determining the worst-case timing behavior of the program, i.e., the path which takes the most time when executed on the hardware platform, is an important problem. Due to non-terminating algorithms, it is not always possible to determine the WCET of arbitrary software. However, in real-time systems, termination is usually a requirement, ensured by specific coding styles, which allows WCET analysis to succeed and compute reasonable upper bounds.

Calculating the WCET of a control algorithm without considering the environment leads to overapproximations which are often too coarse, e.g., when sensor data is assumed from the environment which is in practice impossible.

In this paper, we describe an approach for overapproximating the WCET of an embedded software program considering a platform and an environment model, which describes the observable behavior of the system's environment. The machine code of the program, the platform and the environment model are combined into a timed CPS model that is then analyzed utilizing software model checking techniques. We evaluate the approach in a case study for a thermal controller application executed on a LEON3 SPARC processor and present first experimental results under different environment constraints.

The remainder of the paper is structured as follows: in Section II, we describe the basics of WCET analysis. Section III describes our approach to WCET overapproximation using model checking. In Section IV, a case study is given. Section V presents experimental results for the case study. Section VI concludes.

II. Background

Many approaches for WCET analysis have been proposed, see e.g. [10] for an overview of existing techniques. They can be roughly classified into two categories: dynamic and static timing analysis.

In *dynamic timing analysis*, the execution time of a task is measured for a set of selected test cases executed on the processor, e.g. [9]. The test cases are typically not exhaustive. Therefore, in general, this approach is too optimistic and unsafe as there is no guarantee, that the path exhibiting the WCET is in the set of test cases. Our approach includes finding the worst case inputs which would require an approach based on dynamic timing analysis to search the entire state space of the environment, which is usually not feasible.

In *static timing analysis*, a model of the timing behavior of the processor is constructed and all possible execution paths of the model are examined by means of formal methods. By taking all possible execution paths into account, the result of the analysis will always be safe, i.e., the calculated WCET will always be at least as high as the actual WCET. Without restrictions on the possible inputs, the calculated WCET might be higher than the actual possible WCET, due to the fact that the inputs leading to the calculated WCET might not occur in the real application scenario. This can be mitigated by constraining the possible tasks by means of user supplied annotations. In our approach we annotate the code with a model of the environment behavior, thus enabling tighter bounds on the WCET compared to the unconstrained case.

The authors of [3] reconstruct the control flow graph of the program under analysis, compute possible processor states by means of abstract interpretation and determine the WCET of the basic blocks. The WCET calculation for the whole program is then formulated as an integer linear programming (ILP) problem. In comparison to our approach, this does not take the environment into account, allowing for infeasible paths to increase the WCET overapproximation. Furthermore, we do not calculate the WCET on independent basic blocks but

on the whole program, allowing the WCETs of basic blocks to depend on the execution of other basic blocks.

In [5], the authors propose an approach, where they determine the WCET of the basic blocks of a program, annotate these WCETs in the source code and employ a bounded model checker to determine the WCET iteratively. Compared to our approach, this requires the source code of the program under analysis. Also, no environment model is used to exclude infeasible paths.

[4] formulate the WCET calculation as an optimization problem using satisfyability modulo theory. However, no bounds on the processor registers are taken into account, which in our approach are supplied by the annotation of an environment model.

III. WCET Analysis using Model Checking

In this section, we describe our approach to compute the WCET of embedded software using model checking. Model checking is an algorithmic approach to verify that a system implementation adheres to its specification. In contrast to simulation and testing techniques, model checking guarantees correctness for all possible input scenarios when successful and generates a counterexample, i.e., a detailed execution of the system that exhibits a specification violation, when failing. The counterexamples can then be examined to debug the system implementation. This has shown to be a viable alternative to the usually employed [1] static program analysis and to give tighter WCET estimates [7], [8].

From a user's perspective, the approach takes as input the machine code P of an embedded software program that potentially interacts via sensors and actuators with a physical environment, a platform model $\mathcal{T}$, and a model $\mathcal{E}$ of the environment. The platform model describes the hardware architecture of the target platform including the register file and flags and how it changes when specific instructions are executed; moreover, the timing for all instruction types is described. The environment model describes the entities of the physical world and how they change with actuator inputs. Our approach works in two steps:

1) We construct, as shown in Fig. 1, a timed CPS model $\mathcal{M}$ that combines P, $\mathcal{T}$, and $\mathcal{E}$ by instrumenting P with a global time variable t_{WCET} that is incremented when instructions are executed.
2) We then iteratively model check $\mathcal{M}$, as shown in Fig. 2, with respect to a timing requirement $t_{\mathsf{WCET}} < b_i$, where b_i is the bound on the WCET in the i-th iteration (b_0 is some small value) that is successively increased. In each iteration, the model checker produces a counterexample that corresponds to a path through the program with WCET $t_{\mathsf{WCET}} \geq b_i$. Eventually, in the k-th iteration, $t_{\mathsf{WCET}} < b_k$ cannot be refuted on $\mathcal{M}$ such that $b_{k-1} \leq t_{\mathsf{WCET}} < b_k$ is known to hold for the model. In further iterations the interval $[b_{k-1}, b_k]$ can be narrowed sufficiently to calculate an overapproximation of the WCET, e.g., using a binary search.

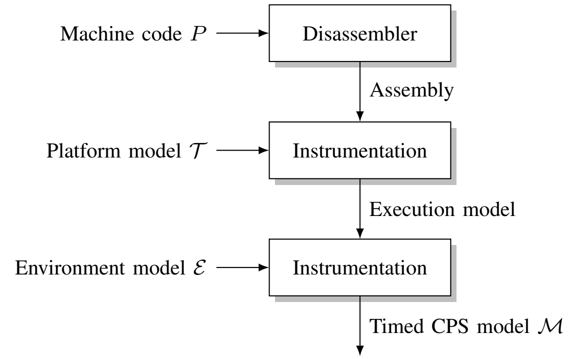

Fig. 1. Construction of the timed CPS model.

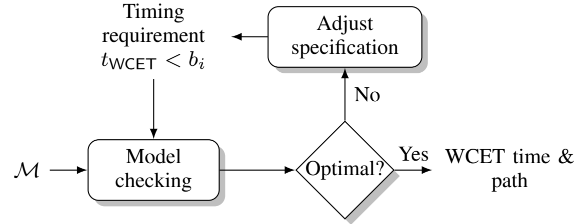

Fig. 2. Overall flow for worst-case execution time analysis.

A. Platform Model

For our approach, a platform model that describes the semantic effects of executing instructions on the target platform is required. This model is manually created for the target platform. We introduce variables for the register file and flags of the processor and describe for each instruction type how those are changed when an instruction is executed. Moreover, a global timing variable t_{WCET} is introduced that keeps track of the execution time; whenever an instruction is executed, t_{WCET} is incremented by the number of clock cycles required by the hardware platform for performing the corresponding computations in hardware. The concrete number of clock cycles to increase t_{WCET} per instruction type are extracted from the processor manual. This model is flexible and can be enriched with more complex timing models to describe effects like caching or branch prediction that are typically implemented in state-of-the-art processors, which is not in the scope of this paper, however.

We model the memory to return nondeterministic values for any memory access, which results in a safe overapproximation and is feasible for realistic systems. Such an inconsistent memory model reduces functional consistency and can enable paths that would not exist in the set of execution traces of the real system. Instructions whose effects are difficult to model were overapproximated by determining the registers or memory locations they modify and setting them to nondeterministic values.

B. Environment Model

Including the environment in the model is achieved by instructing the model checker to assume a memory location

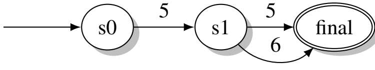

Fig. 3. Example of paths through a program. All paths begin at *s0* and end in *final*. Nodes in between are branching points. The edges represent the basic blocks of the code, where the numbers denote the number of instructions in a basic block.

or register to have a certain value at a certain point in the execution, e.g., when calling a function to obtain a sensor value, its return value can be constrained to a specific value given by the environment model. In the simplest case, such a value is a constant, which means that the WCET's overapproximation is calculated for a certain environment state. However, the WCET in this case could be measured, if the environment state can be reproduced in the current state of development. Generally, any environment model expressible to the model checker can be used to constrain the inputs. In the model, time is implicitly described by the instruction counter, depending on the clock rate of the processor, i.e., the elapsed time between two environment interactions is $\Delta_t = T_{CLK} \cdot \Delta c_{cc}$, where T_{CLK} is the clock period of the system and c_{cc} is the clock cycle or instruction counter. We use this approach to calculate a WCET overapproximation over all feasible environment behaviors and to obtain an environment state in which that WCET is reached. The approach allows to take precautions on the software or system level to avoid such an environment state, if the WCET is too high for the application.

C. Computing WCET using Model Checking

To calculate the WCET for a part of a given binary file, a model as described above is created and subsequently checked using a model checker, similar to [5]. If the model checker does not find a violated property, the execution time the modeled code takes is less than the WCET specified in the model. However, if the model checker finds the execution time to be more than or equal to the WCET specified in the model, it produces a counterexample, showing a path through the model on which the execution time is at least equal to the specified WCET. From this counterexample, the exact state of the model can be derived for every step in the modeled piece of code, including the instruction/cycle counter. Thus, to find the WCET, Alg.1 is used.

Given a model with a corresponding specification as described above this algorithm will modify the WCET stated in the specification and invoke the bounded model checker on the model and specification. If the specified WCET is exceeded by at least one execution path, the bounded model checker will find such an execution path contradicting the specification, a so called counterexample, containing the execution time this path took. If the bounded model checker could not find a counterexample the specified WCET exceeds the actual WCET of the model and is an upper bound on the WCET. The algorithm will iteratively converge on the actual WCET of the model by saving the execution time found in the last counterexample, which is the lower bound found so far, setting

Algorithm 1 WCET

Input: spec, model, maxIterations

1: lowerBound $\leftarrow 0$
2: next $\leftarrow 1$
3: **repeat**
4:   **if** next $=$ lowerBound **then**
5:     **return** next
6:   **else**
7:     WCET (spec) $\leftarrow$ next
8:     counterex. $\leftarrow$ modelChecker (model, spec)
9:     **if** counterex. $=$ none **then**
10:       next $\leftarrow \lfloor \text{lowerBound} + \frac{\text{next} - \text{lowerBound}}{2} \rfloor$
11:     **else**
12:       lowerBound $\leftarrow$ counter (counterex.)
13:       next $\leftarrow$ increment(lowerBound)
14:     **end if**
15:   **end if**
16:   maxIterations $\leftarrow$ maxIterations $- 1$
17: **until** maxIterations $= 0$
18: **return** next

the specifications WCET to a higher value and invoking the model checker again.

To terminate this algorithm even in the case of infinite loops an upper bound on the number of model checker invocations is defined. If the WCET cannot be found in that number of iterations, the value returned is a lower bound of the WCET.

When the model checker encodes the problem as a satisfiability problem, where the formula is satisfied when *next* is lower or equal to the WCET, adjusting *next* by more than a constant factor, e.g., exponentially with the difference to the previous value, would lead to greatly increased runtime. In the case of *next* being greater than the WCET, the given problem is unsatisfiable, which has shown to be much harder to conclude for the SAT solver on the studied models. When defining $increment(x) \equiv x + 1$, this occurs in only one iteration.

Fig. 3 shows an example of a control flow graph, where the number of instructions for a basic block is denoted on the edges. To illustrate Alg.1, we apply it to this program with $increment(x) \equiv x + 1$. In the first iteration, $next = 1$, $next \neq lowerBound$, thus a specification `WCET < 1` (line 7) is model checked on the model of the program (line 8). The model checker determines a counterexample, e.g., $s0 \Rightarrow s1 \Rightarrow final$, with an execution time of 10. Therefore, $lowerBound = 10$, $next = increment(10) = 11$ (lines 12,13). In the next iteration, $next \neq lowerBound$ still holds, thus the specification is modified to `WCET < 11`. Again, the model checker produces a counterexample, e.g., path $s0 \Rightarrow s1 \Rightarrow final$ with an execution time of 11 contradicting `WCET < 11`. Thus $lowerBound = 11$, $next = 12$. In the third iteration, there is no counterexample for `WCET < 12`, therefore $next = \lfloor \frac{next - lowerBound}{2} \rfloor = 11$ (line 10). In the final iteration, $next = lowerBound$ and 11 is returned (line 5).

IV. Case Study

We evaluate our approach in a case study and compute an overapproximation of the WCET of a thermal controller implemented as a embedded real-time software task executed on a LEON3 processor. The software task is a discrete two-point control application that interacts in a closed feedback-loop via three, triple-modular redundant temperature sensors and a heating element with its physical environment. The objective of the thermal controller is to keep the temperature of a battery in a certain interval such that optimal power supply of the system can be guaranteed. We assume that the temperature sensors and the battery are tightly coupled such that all sensors observe the same temperature when function correctly. A majority vote on the sensor values allows that the system tolerates a malfunctioning sensor.

The three temperature sensors and the heating element are connected to the LEON3 procedure in a simple network topology via SpaceWire connections. Thus, the data exchange follows the SpaceWire standard for data communication and wrapped in software through a simplified version of the *remote memory access protocol* (RMAP). When a value is read from one of the sensors or an enable/disable signal is send to the heating element, the data is written to or read from a memory buffer via RMAP API commands. Consequently, neither the sensor nor the heating element are modeled in detail, but abstracted to a set of mapped memory locations.

For the case study, the software controller was implemented in C++ with the real-time operating system RTEMS and compiled with the RTEMS cross compilation system of the GCC cross compiler system to machine code for the SPARC v8 architecture.

A. Obtaining the Timed CPS Model

The assembler instructions were modeled in C. LEON3 is an implementation of the SPARC v8 architecture, which in turn is a *reduced instruction set computer* (RISC) architecture. The instruction set is relatively small and low in complexity, keeping the model relatively simple. On the targeted LEON3 there is no FPU or coprocessor, which reduces the necessary instruction set. Furthermore, no cache is present, making the results more predictable.

Modeling the arithmetic and logic instructions in C is straightforward, as C offers operators for the exact computation of these instructions.

Due to the size of the address space of the LEON3, a 32-bit architecture, it is usually not possible to model the whole memory as array explicitly for this. More generally, for most other common architectures the same problem would arise. Sacrificing functional consistency by making the model for memory storing instructions dummy instructions which only increase the instruction counter and the memory loading instructions write nondeterministic values into the affected registers provides the needed simplification to allow the model to be checked. Furthermore, this is a safe abstraction as it does not exclude the values evoking the actual WCET but rather can increase the overapproximation by enabling paths which would not be feasible if an accurate memory model would be used.

Function calls can be described using `goto`-statements that jump to the address of the function's implementation or, alternatively, using interrupts, or *computed gotos*. In contrast to `goto`-statements that jump to a predetermined, constant address, computed gotos jump to the address of a label value that is an address stored in a variable. In our experiments, we concluded that the first mechanism is more practical. The latter mechanism allows to describe the control flow of a machine code program close to the actual executable, e.g., jumps from inside of one function to an arbitrary location inside of an another function are allowed. However, in our experiments, we found that state-of-the-art model checkers tend to encode computed gotos into a sequence of nested conditional checks (one check per label in the program). The nested checks compare the label of the return address to each label in the program such that this comparison becomes the bottleneck when reasoning about the behavior of larger programs.

B. Translation

The C file is generated in two passes over the assembler file. In the first pass, function definitions and the locations and targets of branch instructions are extracted which are subsequently used in the second pass to identify function calls to functions which are not in the output of the disassembling step, and to insert labels in the places the branch instructions should branch to. Furthermore, to ensure standard compliant C code, which requires all functions to be declared before being called for the first time, all functions should be declared at the beginning of the final C file.

In the second pass, the C file is generated. Therefore, first, all functions found in the first pass are declared, followed by the assembler instructions replaced by the corresponding C macro, wrapped in function bodies. Fig. 4 gives an example of a translated function. When a line is encountered that is the target of a branch instruction, a goto label is generated such that the label can be referenced by the goto statement modeling the branch instruction.

While this will create a model that behaves just as the binary file used as input in many basic cases, in some situations the control flow will differ or be more complex than is the case for a simple C model with functions returning after their last statement. Following situations have been identified where this is the case:

a) Tail call optimization: When the last instruction executed on a certain path in a function is a function call, the compiler might optimize the resulting binary code to let the called function, when returning, return to the function the current function would return to, saving at least one return instruction. Fig. 4 gives an example of such tail call optimizations. The SPARC v8 architecture saves the return address in the `o7` register. Also, `call` is a delay control flow transfer instructions, for which the control flow transfer is not immediate. This means, the instruction following a call instruction is executed while the control flow is transferred

```
40016b30 <fun1>:
40016b30:   ld  [%o0 + 0x34], %g1
40016b34:   cmp  %g1, 1
40016b38:   be  40016b4c
40016b3c:   clr  %o2
40016b40:   mov  %o7, %g1
40016b44:   call  40019adc
40016b48:   mov  %g1, %o7

40016b4c:   mov  %o7, %g1
40016b50:   call  40016b5c
40016b54:   mov  %g1, %o7
40016b58:   nop


40016b5c <fun3>:
40016b5c:   ...
```

```
void fun1() {
    Ld(rO0 + 0x34, rG1);
    Cmp(rG1, 1);
    Be(l40016b4c, Clr(rO2));

    Mov(rO7, rG1);
    Call(fun2(), Mov(rG1, rO7));
    return;
l40016b4c:
    Mov(rO7, rG1);
    Call(fun3(), Mov(rG1, rO7));
    return;
    Nop();

    fun3();
}
```

Fig. 4. Example of the assembler translation.

to the called function. In the example, the return address of the called function is overwritten by the return address of the calling function. By inserting simple C return statements after identified locations of such optimizations, the control flow is adjusted correspondingly without introducing additional instruction cycles.

b) Shared assembler code: Some functions have multiple entry points in the assembler representation, where one entry point introduces a preamble to the following function by not returning before the execution reaches the first statement of the next function. Inserting a call to the next function as the last statement of a function's C representation is a simple solution to this problem simulating the behavior of the assembler description.

V. Experiments

All experiments were conducted on a virtual machine running Linux 4.3.3 with 5270MiB RAM and four processor cores of an Intel Core i5-4590 CPU @ 3.30GHz assigned. We used CBMC 5.3 [2] as model checker.

A. Code Statistics

The analyzed code consists of 27 functions with a call graph depth of 7. Before compilation, there were 216 lines containing C/C++ statements, excluding function signatures. After compilation, the controller code and all code called by that consisted of 586 assembler instructions.

B. Experiment Execution

First, the code is compiled and linked against the required library, which contains the SpaceWire drivers and the memory access protocol stack. Then the approach described in Sections III and IV is applied. After decompiling the binary and translating it to the model to be checked, a WCET overapproximation of the entry function which is the initialization code for the controller and a single control step is determined. Then, the environment model is integrated into the model, constraining the inputs and the WCET is again overapproximated.

We applied the approach to every function in the resulting system. For functions with loops depending on function parameters, safe upper bounds were determined and applied in the process.

The controller will not change the state, i.e., write to the system outputs to change the system and environment state if the measured temperature t fulfills $T_- < t < T_+$. Therefore, constraining the environment to fulfill $T_- < t < T_+$ will make paths in the software infeasible which are taken when the controller turns the heating element on or off, which results in a shorter WCET.

In Tab. I, determining a WCET overapproximation for one step of the controller exceeded a given time bound (denoted as T/O) of 90 minutes after 45 iterations. Therefore, the problem was reduced by assuming functions without input parameters, i.e., independent from their input, to always reach their WCET and return a nondeterministic value, which is a safe overapproximation. By excluding `spwl_reclaim_txbuf` and `spwl_send_txbuf` from the analysis in this manner, the analysis of `TemperatureController::controlStep` finished after 49 iterations in $178.52s$, resulting in a WCET of 2634 instructions. Constraining the read sensor values to the interval (T_-, T_+) by changing the environment model appropriately resulted in a WCET of 1995 instructions after 37 iterations in $57.53s$ which is a significant difference both in calculated WCET and algorithm run time. The latter can be explained by the fact that the tighter constraints provided by the environment reduce the state space. Further experiments extending the interval in either direction show, that the case $t > T_-$ has a lower WCET than $t < T_+$, which was to be expected. The latter case includes values where the heating is turned on, which means `Heating::setOn` is called which in turn has a higher WCET than `Heating::setOff`.

VI. Conclusion

We proposed an approach for overapproximating the WCET of embedded control software using model checking. The approach works in two steps: first, a timed CPS model is derived from a software program, a platform model, and an environment model. Second, a timing requirement is model

TABLE I

TABLE SHOWING THE PERFORMANCE AND RESULTS OF THE WCET APPROACH APPLIED TO THE FUNCTIONS OF THE MODEL. *D*: DEPTH OF SUBCALLGRAPH. *C* FUNCTIONS CALLED FROM THE FUNCTION. #INSTR: ASSEMBLER INSTRUCTIONS THE FUNCTION CONSISTS OF. TIME: ALGORITHM RUN TIME. #ITER: NUMBER OF ALGORITHM ITERATIONS. WCET: WCET OF THE FUNCTION. †: WCET INCLUDES SOFTWARE TRAP HANDLING CODE.

Function	*D*	*C*	#Instr.	Time	#Iter.	WCET (#instr.)
obc::Crc8::calculate	0	0	15	16.75	51	232
obc::amap::Amap::Amap	0	0	11	0.51	3	11
obc::leon3::SpaceWireLight::flushReceiveBuffer	0	0	2	0.51	3	2
obc::leon3::SpaceWireLight::close	0	0	2	0.51	3	2
obc::leon3::SpaceWireLight::SpaceWireLight	0	0	10	0.52	3	10
obc::leon3::SpaceWireLight::~SpaceWireLight	0	0	6	0.5	3	6
reap_tx_descriptors	0	0	39	1.11	5	39
sparc_disable_interrupts †	0	0	4	0.52	3	17
sparc_enable_interrupts †	0	0	4	0.52	3	21
TemperatureSensor::TemperatureSensor	1	2	13	0.51	3	32
TemperatureSensor::~TemperatureSensor	1	2	8	0.51	3	14
obc::amap::Amap::writeHeader	1	1	24	2.8	11	95
Heating::Heating	1	2	15	0.52	3	36
Heating::~Heating	1	2	8	0.51	3	14
spwl_reclaim_txbuf	1	3	54	6.7	20	169
spwl_send_txbuf	1	3	94	6.9	18	197
TemperatureController::TemperatureController	2	2	20	0.53	3	151
obc::leon3::SpaceWireLight::requestBuffer	2	1	23	6.02	17	188
obc::leon3::SpaceWireLight::send	2	1	41	6.92	12	226
obc::amap::Amap::read2	3	4	40	631.78	166	1308
obc::amap::Amap::write2	3	5	50	1044.3	190	1005
obc::amap::Amap::read1	4	1	9	1041.8	190	1005
obc::amap::Amap::write1	4	1	9	1159.8	188	1014
TemperatureSensor::readValue	5	1	11	182.17	51	656
Heating::setOn	5	1	19	88.39	35	638
Heating::setOff	5	1	17	96.04	33	636
TemperatureController::controlStep	6	3	38	T/O	45	> 2555
TemperatureController::controlStep	6	3	38	178.52	49	2634
TemperatureController::controlStep $(T_- < t < T_+)$	6	3	38	57.53	37	1995
TemperatureController::controlStep $(T_- < t)$	6	3	38	177.76	50	2630
TemperatureController::controlStep $(t < T_+)$	6	3	38	119.55	43	2634

checked on the timed CPS model and adjusted until the WCET execution path is determined.

Compared to similar state-of-the-art approaches on WCET overapproximation [5], [6], deriving the model from a binary image of the program allows the application to more realistic scenarios where an optimizing compiler is used. Furthermore, we contribute the addition of an environment model, constraining the behavior of the system to actual use cases, which to our knowledge has not been done before. Deriving the WCET from the binary image under the environment constraints allows for tighter upper bounds, i.e., less overapproximation. We conducted a case study with a thermal control application and presented first experiments for simple environment models. We conclude that our results are promising and using the environment models significantly reduces the overapproximation of the WCET. As future work we consider implementing our approach directly a satisfiability modulo theory solver (SMT solver), e.g., similar to [4] and conduct further case studies.

REFERENCES

[1] Adrish Banerjee, Subrata Chattopadhyay, and Abhik Roychoudhury. Precise micro-architectural modeling for wcet analysis via ai+sat. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 87–96. IEEE, 2013.

[2] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988, pages 168–176, 2004.

[3] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

[4] Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maïza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. *ACM SIGPLAN Notices*, 49(5):43–52, 2014.

[5] Sungjun Kim, Hiren D Patel, and Stephen A Edwards. Using a model checker to determine worst-case execution time. 2009.

[6] Matthew MY Kuo, Li Hsien Yoong, Sidharta Andalam, and Partha S Roop. Determining the worst-case reaction time of iec 61499 function blocks. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 1104–1109. IEEE, 2010.

[7] Alexander Metzner. Why model checking can improve wcet analysis. In *Computer Aided Verification*, pages 334–347. Springer, 2004.

[8] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Accurate and efficient identification of worst-case execution time for multicore processors: A survey. In *Design and Test Symposium (IDT), 2013 8th International*, pages 1–6. IEEE, 2013.

[9] Sanjit A Seshia and Alexander Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):55, 2012.

[10] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem — Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.