Adaptable and Precise Worst Case Execution Time Estimation Tool

Vladimir-Alexandru Paun, Bruno Monsuez

Adaptable and Precise Worst Case Execution Time Estimation Tool

Vladimir-Alexandru Paun
UEI, ENSTA ParisTech
Paris, France
paun@ensta-paristech.fr

Bruno Monsuez
UEI, ENSTA ParisTech
Paris, France
monsuez@ensta-paristech.fr

Abstract

Real-time systems are everyware. When they are integrated into safety-critical systems, the verification of their properties becomes a crucial part. Besides the growth in complexity of the embedded systems, platforms are getting more and more heterogeneous. Being able to validate their non-functional properties is a complex and resource consuming task. One of the main reasons is that currently available solutions focus on delivering precise estimation through tools that are highly dependent on the underlying platform as in order to provide precise and safe results, the architecture of the system must be take into account. In this project we address these issues by developing a prototype that maintains a good level of precision while being adaptable to a variety of platforms by separating as much as possible the worst case execution time estimation stage from the hardware modeling aspects.

General Terms Hard Real-Time Systems, precision, safety, adaptability

Keywords WCET, Abstract State Machine, Symbolic Execution

1. Introduction

With regard to the respect of the timing constraints, real-time systems are classified in two categories: hard real-time systems (the non respect of a deadline can lead to catastrophic consequences) and soft real-time systems (missing a deadline can cause performance degradation and material loss). We analyze hard real-time systems that need precise and safe determination of the worst case execution time (WCET) bounds that are crucial in the certification process. Traditionally two approaches are used, namely dynamic and static methods [1], we only consider the latest as dynamic methods, in the traditional sens, fail to deliver safe estimations for modern platforms that contain, for example, pipelines or cache memories and tend to greatly underestimate the WCET.

In order to give a safe estimation of the WCET, all the interactions and reachable states of the system must be analyzed or over approximated, hence the need of an analysis that takes into account the exact underlying architecture. We choose to separate as much as possible the modeling part from the analysis part in order to achieve the flexibility needed to adapt to new hardware.

In our approach we start from the system's model and the binary that will be executed on the final platform. An extension of the Symbolic Execution (SE) [2], the *conjoint* SE, will generate all the reachable states of the processor, under the supervision of a *prediction module* that will *fusion* identical and similar states in order to contain the state space explosion and give details regarding the global precision loss of the WCET estimation.

In the following we first take a look into the state of the art concerning timing analysis and we continue with the description of the high level architecture of our tool. Subsequently we take a closer look into the formal model used to simulate the hardware that gives us the edge in the adaptability of our tool followed by a presentation of the WCET estimation steps and the transformations needed to contain the combinatorial explosion.

2. Related works

Many of the available time analysis tools show a list of compatible hardware and present each new platform taken into account as a new feature. One of existing methods, OTAWA, introduced by Cass and Sainrat [4], differentiates itself by making a first step towards adaptability as it uses a parametrized model of a generic platform that can address a variety of architectures but nevertheless, the process is fairly difficult and the model lacks precision while it fails to capture the precise behavior of the platform. AbsInt, one of the

leading WCET analyzers is also taking a step towards adaptability by apparently looking to use a SystemC description in order to generate an abstract model. The main issue is that the SystemC language has only recently come to be a standard therefore descriptions of older hardware (mainly used in hard real-time systems) are not common. To our knowledge non of the other WCET tools have embraced the adaptability paradigm.

3. The global architecture of the WCET estimation tool

The two main entries of the tool are the processor model and the program binary, as depicted in Figure 1. The processor is regarded as the union of it's components $\mu P = \bigcup_{i=0}^{n} C_i$ and modeled as a hierarchical timed abstract state machine, described further in the paper, that has the useful feature of enabling multiple definitions for a same component C_i. A supervisor that we call the *Oracle* decides what abstraction level is best suited for the current context in order to optimize the *precision* to *state explosion* ratio. An external value analyzer is used to obtain information regarding the instruction order, their addresses and the control flow graph of the program. Symbolic execution is used to symbolically execute each instruction of the program, meaning that each variable has initially a symbolic value (as we generally do not posses exact information on it's value) that gets refined by accumulating all the informations and decisions taken during execution. One of the advantages of this method is that it manages to simulate the interactions inside the processor in detail, for example capturing by construction the timing anomalies [5]. The *SE* generates all reachable states of the processor, meaning that we have to manage a rapidly increasing state space. Our fusion stage consists in merging as much states as possible without affecting to much the precision of the estimation. We achieve this by using the prediction module that will first identify the states that are good candidates for merging and then estimate the impact of the fusion on the global analysis. After browsing and evaluating the processor's states, the time corresponding to the worst path is selected.

4. Timed Hierarchical Abstract State Machines

4.1 Abstract State Machine Formalism

The sequential ASM Thesis, introduced in [6] proves the isomorphic modeling of any algorithm. The sequential ASM algorithm consists of a set of *rules* applied to *states* in a sequence of steps assimilated to a *run*. States are structures in the sense of first-order logic, with relations trated as Boolean-valued functions. A finite collection of function names having a fixed arity is called a *vocabulary*, Γ. A state S of vocabulary Γ is a non-empty set X, together with the interpretation of all function names in Γ over X, therefore holding the values of all the variables at a specific step. *Up-*

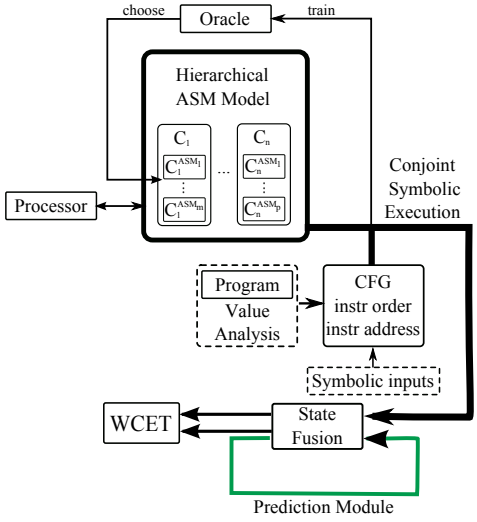

Figure 1. Global architecture of the WCET estimation tool

dates represent the simplest change that can occur to a state by the change of the interpretation of a function at one particular tuple of arguments. Let R be a rule that gives rise to a set of updates. In order to execute R at S all the updates are triggered in the corresponding update set. Thus we have the *update rule*, the *block rule*, a sequence of transitions rules that are executed simultaneously, the conditional rule **if** g **then** R_0 **else** R_1 **endif**, etc.

4.2 Hierarchical Timed ASM

Possessing a precise and versatile model of the processor is very important. Nevertheless having access to an usable HDL code, is rarely the case for platforms used in hard real-time systems, that are fairly outdated, and even if it exists, their is no common, unified description language. Ideally we should use the description of the processor as an input and generate an usable model for the analysis. As the lack of availability and standardization makes the task impossible, the need to create a model for each platform is mandatory. This is one of the bottlenecks in the adaptability of current tools, and we consider that the modeling part should be therefore an separated straightforward engineering task that can be made on the fly and without disposing of precise knowledge with regard to the rest of the tool. Therefore we chose to use the abstract state machine, a method that bridges the gap between human understanding and formulation of real-world problems and the deployment of their algorithmic solutions, in our case, the modeling of the processor, that showed it's efficiency as a specification method in numerous practical applications (e.g. see [7], [10]).

Using a human readable and machine executable language makes the difference when it comes to speeding up the process of the hardware description. However some important features were not included in the original version of the ASMs [6] like the timing aspects hence updates are consid-

ered immediate. Ouimet et all. [8] introduced the concept of durative actions by adding delays directly in the syntax; our approach is similar. In [9] a prototype of a simulator for reactive timed ASMs that verifies the respect of requirements specifications. Besides the timing aspects we enrich the original model with hierarchical feature that enables us to give different definitions on several abstraction levels of the same processor component.

The goal of hierarchical ASMs is to provide at any time during the analysis, the right level of abstraction in order to prevent the combinatorial explosion. We know that we do not always dispose of precise information during the analysis (e.g. data memory address, availability in the cache, etc.) therefore using the most precise description of the fetching mechanism, for example, would be useless, on the other hand, a less precise, more abstract, definition could help reduce the number of generated states.

The hierarchical definition of components integrates seamlessly into the ASM formalism. Basically, the *oracle* is an ASM module that imports all the needed function definitions and exports the needed functions or rules. Each hierarchical module is defined as a control state ASM (cf. [10]) using in it's condition the result from the *oracle* that decides which implementation is appropriate for the current context.

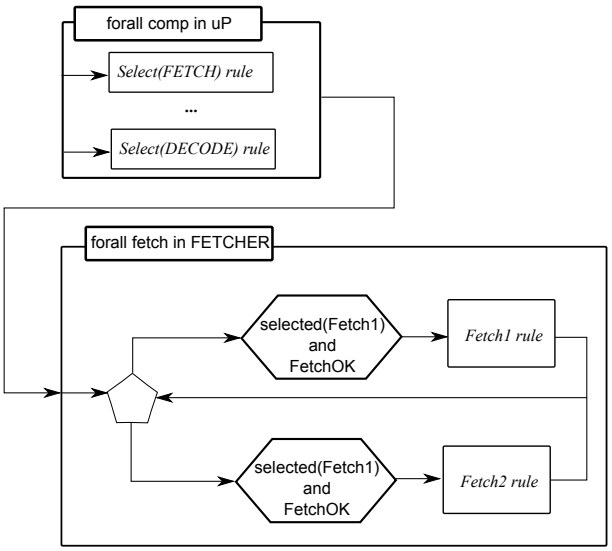

Figure 2. The oracle and the fetcher modules

FETCH =
forall fetch ∈ FETCHER **do** FETCH1(fetch), FETCH2(fetch)

In Figure 3 we have two definition of the Fetch stage, the first one corresponding to the more abstract version that will typically be chosen if we have no precise information on the exact fetch address. Generally we have a family of abstraction for each component of the processor, $\alpha_{C_i} = \bigcup_{j=0}^{m} \alpha_j$ so that $C_i \overset{\alpha_j}{\rightarrow} C_i^{\alpha_j}$. Let $T(C_i^{\alpha_j})$ be the contribution

```
FETCH
if FetchOK then
    FetchQueue:=getNextInstr()
    t:+=[t_min, t_max]
endif

if FetchOK then
   FetchAddr:=getExactFetchAddr()
   howMany:=FetchAddr MOD 4
   FetchQueue:=BurstAccess(
            FetchAddr,howMany)
   t:+=[t_BurstFetch]
endif
```

Figure 3. Different definitions of the fetcher

of the abstract component to the global execution time. We must have $T(C_i^{\alpha_j}) \sqsupseteq T(C_i)$.

5. Conjoint Symbolic Execution

The use of *SE* to analyze the intra-processor interactions has been used with good results in [3], however the method suffers from the lack of a precise hardware model and inaccurate merging strategies that lead to important overestimations. The basic *SE* consists in replacing the variables with symbolic values and extending the operations in order to take this into account. The interpretation of the assignment rule is straightforward. Let $p(pc)$ be Q, $p(x_i)$ be E_i and $p(\alpha \leftarrow \beta)$ be the old p where the value of α is changed to β. A special treatment is applied to conditional instructions that use the pc to explore all the possible scenarios. The expressions conjoined in the pc are of form $Q > 0$ where Q is a polynomial over symbolic values. Let R be this expression we thus have three possible cases: we can determine from the pc that the condition is always true ($pc \supset R$ and $pc \not\supset \neg R$), analogue for always false or we can not determine if the condition is true or false, $pc \supset R$ and $pc \supset \neg R$, therefore the execution will continue along both branches, generating two new paths.

The first step of our conjoint *SE* deals with the program's CFG that is regarded as an input for the processor's model *SE*.

6. Smart State Fusion

One of the major drawbacks of the SE comes from its quality of generating every feasible path, that for a real-life industrial program generates a combinatorial explosion that is not obviously containable. What still remains challenging today is to handle this explosion while still remaining precise enough. This translates to finding a way of eliminating some of the states, and we choose the technique of states fusion that will try to generate an abstract state capable of

capturing the respective states features, with regards to the goal, but remain as compact as possible. It has been proven in [11] that because of the finite number of states that a processor can have and because of the constrains generated by the execution contexts at a certain point we will have states that regardless of the different history, will generate identical or very similar new states. One major step in having precise fusions is to determine when to make them and what changes to apply. States can be of two types: identical, meaning that they have either all the elements that are the same, in this case we can suppose that an eventual fusion will not impact the precision of the analysis, or similar, some of the components are not the same so we proceed to another analysis to determine to which extent they are different. Therefore similar states can be strongly or weakly similar, meaning that the impact of the fusion will be acceptable or not. For the instant this estimation is done dynamically by our prediction module. Its goal is to evaluate the impact in the future of a fusion by unrolling the tree for several steps (generally equal to the pipeline depth), continuing the execution along the paths before and after fusion and comparing the result. Further details about this technique can be found in [11].

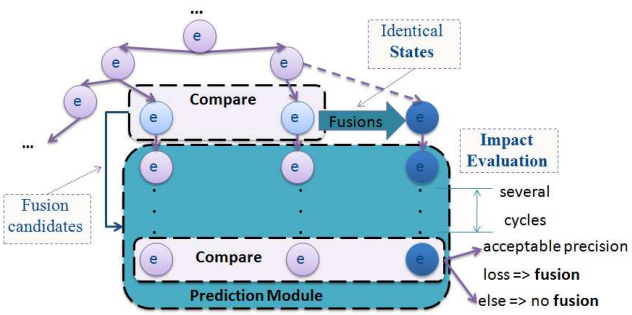

Figure 4. The Dynamic Fusion - snapshot of the Prediction Module

7. Global algorithm

1. Start from the initial state: where all the components have the unknown value and pc is set to $true$
2. For every variable that we encounter and that we do not have the exact value, assign a symbolic value
3. Activate the first ASM model and then add the guard condition g to the pc
4. Choose from the *oracle* the appropriate version of the ASM modules
5. Compute the update set of the current step
6. Apply the update set (taking into account that some terms will have symbolic values)
7. Add the result of the update set to the global system state
8. Add the generated states to the collection of next states to be executed
9. Add the duration of the transition to the global time
10. Repeat from point 2. until the collection of next states is empty

8. Conclusions

The world of embedded software is no longer integrating simple hardware/software therefore critical systems are becoming more and more difficult to prove and certify. The growth in complexity and variety increases the need of versatile analyze methods and adapted tools, that can easily and as costless as possible deal with a large panel of architectures. To this end we presented a novel approach that is able to respond to the evergrowing demands and to place itself into a real industrial context.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom The Worst-Case Execution Time Problem Overview of Methods and Survey of Tools, *ACM Transactions on Embedded Computing Systems (TECS)*, Volume 7, Issue 3, April 2008.

[2] T. Lundqvist and P. Stenstrom, An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution, in *Real-Time Systems*, Volume 17, 183-207, November 1999.

[3] T. Lundqvist, A WCET Analysis Method for Pipelined Microprocessors with Cache Memories, Goteborg, Sweeden, 2002.

[4] H. Cass and P. Sainrat, Otawa, A framework for experimenting WCET computations, *ERTS06*, 2006.

[5] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, A vDefinition and Classification of Timing Anomalies, *WCET06*, 2006.

[6] Yuri Gurevich, Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods, ed. E. Brger, *Oxford University Press*, 1995, 9–36.

[7] University of Michigan, ASM homepage. http://www.eecs.umich.edu/gasm/.

[8] M. Ouimet and K. Lundqvist, The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering, *JUCS*, 2007.

[9] A. Slissenko and P. Vasilyev, Simulation of Timed Abstract State Machines with Predicate Logic Model-Checking, *JUCS*, 2008.

[10] E. Borger and R. Stark, Abstract State Machines: A Method for High-Level System Design and Analysis, *Springer-Verlag*, 2003.

[11] Bilel Benhamamouch, Bruno Monsuez: Computing worst case execution time (wcet) by symbolically executing a time-accurate hardware model (extented version), *International Journal of Design, Analysis and Tools for Circuits and Systems*, Volume 1, No. 1, November 2009.