Proceedings of the International MultiConference of Engineers and Computer Scientists 2009 Vol II
IMECS 2009, March 18 - 20, 2009, Hong Kong

Computing worst case execution time (WCET) by Symbolically Executing a time-accurate Hardware Model

Bilel Benhamamouch, Bruno Monsuez *

Abstract—**To ensure that a program will respect all its timing constraints we must be able to compute a safe estimation of its worst case execution time (WCET). However with the increasing sophistication of the processors, computing a precise estimation of the WCET becomes very difficult. In this paper, we propose a novel formal method to compute a precise estimation of the WCET that can be easily parameterized by the hardware architecture. Assuming that there exists an executable timed model of the hardware, we first use symbolic execution [1] to precisely infer the execution time for a given instruction flow. Then we merge the states relying on the loss of precision we are ready to accept.**

Keywords: static analysis, WCET, processor modelization, symbolic execution.

1 Introduction

Computing WCET is useful either to determine appropriate scheduling schemes for the tasks or to perform an overall schedulability analysis. This is done obviously in order to guarantee that all timing constraints will be met. Computing program execution time has always been difficult. Dynamic methods [11, 12] as well as formal methods [5, 8] have received a lot of attention for precise estimation of the worst case execution time of code snippets. However, current methods have some difficulties to cope with the increasing complexity of the hardware used to implement critical real time applications.

In this paper, we present part of our formal method [15] to compute an upper estimation of the WCET that contains the loss of precision and that can also be parameterized by current complex hardware architecture like super-scalar microprocessor or multi-processor systems.

The contribution of this paper is twofold, on one hand we show how easily we abstract the behavior of a current hardware architecture in order to highlight its timing properties, and on the other we propose an approach which uses this abstraction to compute the WCET of a sequence of code.

The paper is organized as follows: we first present how the formal methods which are usually used to compute WCET (Section 2) and we give an overview of our framework (Section 3). Then, we show how we compute the execution time of an instruction sequence using symbolic execution as well as how we abstract the executable timed model to manage the explosion of the states generated by the symbolic execution (Section 4). After that, we illustrate our approach through an example (Section 5). In Section 6, we give an overview of how the merging policy works. Finally, we conclude and we present ongoing and future works (Section 7).

*Ecole Nationale Supérieure de Techniques Avancées UEI, ENSTA 32 Bd Victor, 75739 Paris cedex 15, France Bilel.Benhamamouch@ensta.fr, Bruno.Monsuez@ensta.fr

2 Formal methods

Nowadays the most mature method which allows to compute an over approximation of the WCET is the one developed by AbsInt team. This method can be described as follow:
Initially, a control flow graph (CFG) is extracted from the binary code. Then on this CFG a value analysis is carried out to produce an approximation (intervals) of the memory areas which will be reached. This result is in turn exploited by the following stage represented by the cache analysis which is used to classify the memory references in:

- Cache always hit: The memory reference always results in a cache hit.
- Cache always miss: The memory reference always results in a cache miss.
- Persist: The referenced memory block will be load at most once.
- Not classified: The memory reference could not be classified in one of the above groups.

When this classification is made for all blocks, it is injected as input of the following analysis. This will define the possible states of the pipeline at each execution point of the analyzed program. Therefore, it will be associated to each instruction various execution times (each one is related to a precise environment). At this moment it is useful to specify that the implementation of these various analyses was possible because it is based on abstract interpretation [9, 10]. This technique is largely used for program checking, and it makes possible to associate concrete values to abstract ones (a whole of concrete values could thus be represented by an abstract value). The various results obtained during the preceding stages, are finally exploited jointly with the source code by the last analysis called path analysis. This analysis is based on linear programming techniques. That enables it to produce the longest execution path.

This approach is represented by a sequential analysis formed by black boxes [5, 8]. This is one of the strong points of this approach, considering that it makes possible to use different

techniques at various levels of the study, such as: using abstract interpretation for the cache and the pipeline analysis, while the path analysis is done by ILP (integer linear programming). But this strength can also be a weakness [6, 7]. Indeed, the increase in complexity of the hardware platform leads to an increase in the number of black boxes required to perform the analysis as well as a more complex design for each black box that abstracts the hardware semantics.

In addition, those formal methods have three main drawbacks. During the analysis the dependencies between the black boxes cannot be identified precisely which leads those methods to explore a superset of all execution paths. Thus WCETs for unfeasible execution paths are taken into account (1); To avoid the state explosion, execution paths are merged using tools provided by abstract interpretation like widening operator, that may also conducts to an over-exaggerated approximation of the execution time (2); The analyzer must explicitly support the target platform and must provide valuable abstraction of the hardware components that compose the target platform (3).

3 Our framework

Our approach is mainly based on the symbolic execution. So before explaining how the method works let us describe what is the symbolic execution.

3.1 Background: Symbolic execution

The main idea behind symbolic execution [1, 2] is to use symbolic values instead of actual data to represent the input values. As a result, the output values computed by a program are expressed as a function of symbolic values. Evaluation of assignments is done naturally, the left-hand side variable receives the resulting symbolic expression, which should be a polynomial.

Evaluation of alternatives is a bit more complicated. It requires that a "path condition" PC – a Boolean expression over the symbolic inputs – is added to the execution state. The path condition PC is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated execution path. At program start, each symbolic execution begins with PC initialized to true. When encountering an alternative, evaluation first starts with the evaluation of the associated Boolean expression by replacing variables by their values. Since the values of variables are polynomials over the symbols, the condition is an expression of the form: $P > 0$, where P is a polynomial. Call such an expression R. Then we can have three cases:

- $\mathsf{PC} \supset R$ and $\mathsf{PC} \not\supset \neg R$: The expression is always true, the execution continues with the conditional code sequence.
- $\mathsf{PC} \supset \neg R$ and $\mathsf{PC} \not\supset R$: The expression is always false, the execution continues with the "else" code sequence if an "else" block is available or simply ignore the conditional code sequence.
- Otherwise, the Boolean condition may be true or false. In this case, we split the path condition in two paths conditions $\mathsf{PC}_{\text{true}} = \mathsf{PC} \wedge R$ and $\mathsf{PC}_{\text{false}} = \mathsf{PC} \wedge \neg R$. We continue the concurrent execution of the condition code sequence with $\mathsf{PC}_{\text{true}}$ and the "else" code sequence or the code located after the conditional code sequence with the path condition $\mathsf{PC}_{\text{false}}$.

3.2 Conjoint symbolic execution of binary code and time-accurate system model

To mitigate the drawbacks of the formal methods (section II), we propose a new approach that extends the classical frameworks for computing the worst case execution time of a sequence of code with no loops or branch instruction. This new framework provides two main advantages over the methods currently used: (1) it simply requires an executable timed-model of the target platform and does not require the design of black boxes that abstract the hardware semantics, this is achieved by *the conjoint symbolic execution of the program code and the executable model of the processor*, (2) it provides a method that allows to identify execution states that can be merged with no loss of precision as well as gives insight in the resulting loss of precision when merging execution paths that have similar but different execution times, this is achieved by *the backward execution paths merging with symbolic execution lookup policy*.

In this paper, we focus on the conjoint symbolic execution part of the analysis method. During symbolic program execution, the executable model of the processor is used to compute for each execution point all the states that the processor may reach when executing this instruction with respect to the execution history. Following this reasoning we build a symbolic tree which contains:

(1) All the states that the processor may reach during the execution (only the possible paths are taken into account).

(2) The transitions are labeled with the maximum number of clock cycles needed to move from one state to another.

So at the end of the analysis, we easily extract from the tree the temporally longest execution path. In the next section (Section 4), we explain how we build a symbolic tree which contains all the possible execution times of the analyzed program as well as how we abstract it in order to reduce the number of the generated states whithout introducing any loss of precision. Then, we present some significant architectural details

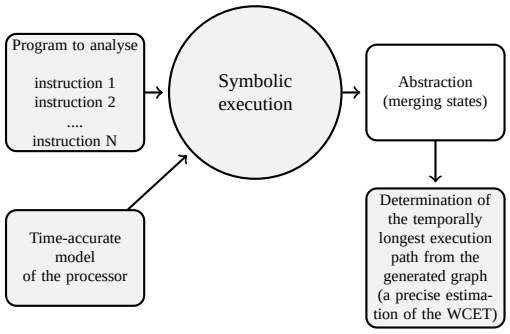

Figure 1: WCET estimation method

of the hardware used in our framework as well as a small program to illustrate our approach (Section 5). Finally, we give in section 6 an overview of how the merging policy works.

4 Using an executable timed-model of the target system to Compute the WCET

4.1 An executable timed model of the target architecture

During our analysis we must provide an executable timed model of the hardware. This abstract timed model is first modeled in C++ with some SystemC facilities. Further extensions could accept full SystemC TLM-T, SystemC Verilog, VHDL or Verilog descriptions.

The executable timed model Basically a processor can be seen as a complex component which is composed by several units. Each one carries out a number of tasks during a clock cycle. The current processor state is the product of the states of all the basic units of the processor.

Definition 1 ***System unit states & system states*** *A system unit state $\mathcal{SC}[u]$ is a minimal set of properties that allows to define what is the next operation that this unit u will perform.*
The state of the target system $\mathcal{SC}$ is the product of all the states of the units that compose this system $\mathcal{SC}$: $\mathcal{SC} = (\otimes_{u \text{ unit of } \mathcal{S}} \mathcal{SC}[u])$.

4.2 Building a symbolic graph

The state s of a symbolically executed binary program includes the system state $\mathcal{SC}$ the processor state (pipeline, data/instruction cache and the PC), and SV the symbolic values. The executable timed model is symbolically executed for the given program. This model takes as an input a current state $\mathcal{SC}$ of the system and returns the processor states $\mathcal{SC}'$ obtained after executing the model during one clock cycle.
We begin the symbolic execution of the code snippet with an empty pipeline ($\mathbf{P} = \mathbf{empty}$) [1] and no information about the cache states ($\mathbf{IC} = \mathbf{DC} = \top$).
Starting from this initial state and relying on the processor description we execute the code symbolically. So, we compute on each clock cycle the set of next states. This set contains all the states that the processor may reach when it starts executing the code with respect to the execution history. For instance, after each cache miss the processor initiates a memory transaction that loads a cache line. If during the execution a cache miss occurs when accessing a word, the cache gets updated. So accessing the double word that follows immediately the loaded word will result in a cache hit.

4.3 Simplifying the generated symbolic graph

The model of the target architecture must be timed, that means that it must preserve the time (number of clock cycles) the processor needs to execute an instruction. The simplest implementation of timed model use the clock as the base cycle. So for each clock cycle, it computes the new state of the processor. However, in the presence of cache misses and pipeline stalls, it may lead to unnecessary intermediate states, since the processor is waiting for some data. A more efficient implementation is the time-accurate model, as shown on figure 2. It is achieved when returning the next processor state that is different from the current one as well as the number of clock cycles required to reach this processor state.

Definition 2 ***Clock-accurate model & Time-accurate model*** *An executable* clock-accurate model *is an executable function that maps a system state $\mathcal{SC}$ to the next system state $\mathcal{SC}'$ at the next clock cycle. An executable* time-accurate model *is an executable function that maps a system state $\mathcal{SC}$ to the pair of a system state $\mathcal{SC}'$ and the time $t \in \mathcal{T}$ needed to reach this system state.*
The atomic times that are associated to a particular cache operation are:
th: time associated to a cache hit.
tm: time associated to a cache miss.
trl: time associated to a cache line reloading.
(d): data cache. **(i)**: instruction cache.
During the execution data and instruction caches are abstracted to indicate the state of the cache (busy or idle) as well as to tell if the requested data is present in the cache.

4.4 Symbolic execution of the time-accurate model

As described before, the time-accurate model takes as an input a current state $\mathcal{SC}$ of the system and returns the next processor state $\mathcal{SC}'$ that is different from the current one as well as the number of clock cycles required to reach this system state. Symbolic execution of the time-accurate model takes as an entry the current symbolic state s of the system and returns a set final states as well as the respective times required to reach this final states: $\{(s_1, t_1), \ldots, (s_n, t_n)\}$.

Definition 3 ***Intermediate states*** *If $s \in \mathsf{S}$ is a valid system state, we call "intermediate states" the final states $\{(s_1, t_1), \ldots, (s_n, t_n)\}$ generated by one step of symbolic execution of the time-accurate model when starting the execution with the state s.*

Definition 4 ***Timed Symbolic Execution Graph*** *The symbolic execution graph $(\mathcal{N}, TR, \mathcal{M})$ is a graph that describes the symbolic execution of a code sequence and is defined as follow:*

- *the nodes $\mathcal{N}$ of the graphs are symbolic states,*
- *the transitions $TR : (\mathcal{N} \times \mathcal{N})$ map a node $\mathcal{N}$ to another node $\mathcal{N}$.*
- *the labeling function $\mathcal{M}$ maps $TR \longrightarrow \mathcal{T}$ and associates to each transition $tr = (n_s \times n_e)$ the number of clock cycles that the system takes to go from the starting state n_s to the ending state n_e of the transition.*

Now, we present the algorithm that builds the *timed symbolic execution tree* for a sequence of binary instructions.

[1] This assumption is made because starting an execution with an unknown pipeline state implies having some information about the running application, which implies other assumptions.

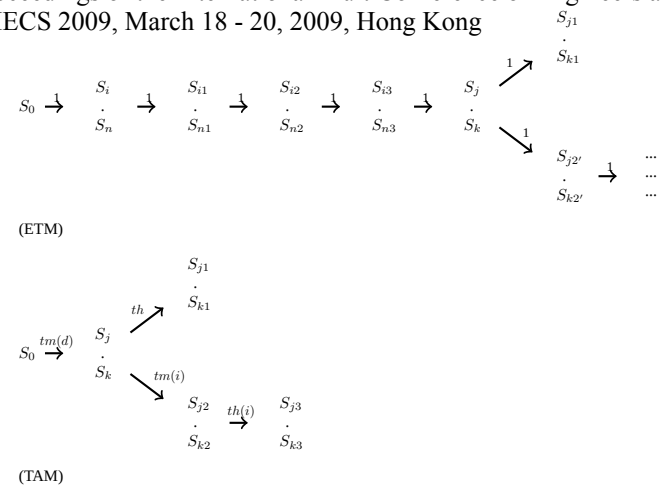

Figure 2: Symbolic execution of the executable timed (ETM) and the time-accurate model (TAM)

$r \leftarrow \{$ initial processor state (pipeline & data cache), `PC` = `true` $\}$, ; // **Initialization**
`current_states` $\leftarrow \{r\}, \mathcal{G} \leftarrow \{r\}$;
while `current_states` *is not empty* **do** // **Propagation**
Removes state s from `current_states`;
Computes the symbolic successors $\{(s_1^s, t_1), \ldots, (s_n^s, t_n)\}$ of s;
Adds all transitions $s \longrightarrow^{t_i} s_i^s$ to the symbolic execution graph $\mathcal{G}$;
Adds to the set `current_states` all the states s_i^s that are not terminal states (terminal states are the final states generated by the last instruction of the code sequence);

5 Framework's illustration

In this section we propose a description of a hardware architecture in order to show how we abstract this description and how we use the resulting abstraction to compute a precise estimation of the WCET. Before starting the processor's modelization, we should decide about the characteristics that we will need during the analysis. These characteristics represent, on each clock cycle, not only the state of the processor but also allow to identify precisely what are the reachable states on the next clock cycle.

5.1 Processor's description

To build the processor state, first we can imagine that a combination of the pipeline and the cache states will suffice to have a precise representation of the hardware. However, during the analysis, relying on this representation we cannot identify precisely the next reachable states. To solve this problem, the processor's state must be enriched with a "path condition (PC)" (Section 3). This PC provides to identify at each execution point of the control flow graph which path should be taken.
So we represent a processor state by a combination of:
(1) a pipeline state (see below the definition).
(2) cache state (instruction and data): it is represented by the instruction (or data) that it contains.
(3) a PC: as seen in section 3, it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated execution path.
Now we will focus on describing the pipeline state. Note that most of the pipelines developed in order to be integrated into an embedded system contain at least :
Fetch: On each clock cycle, this unit retrieves instruction(s) from the memory system and computes the location of the next instruction(s).
Dispatch: This unit decodes the instructions supplied by the instruction fetch stage and dispatches them to the appropriate execution unit.
Execute: Each execution unit that has an executable instruction executes the selected instruction, and notifies the completion stage that the instruction has finished execution. Nowadays, most of the pipelines integrate at least five execution units : an integer unit (IU), a floating-point unit (FPU), a branch processing unit (BPU), a load/store unit (LSU), and a system register unit (SRU).
Complete/writeback: This pipeline stage maintains the correct architectural machine state and transfers the results to the appropriate registers as instructions are retired.
The following table summarizes the pipeline stages.

Processor units	
F: Fetcher	**CU**: Completion Unit
D: Dispatcher	**SRU**: System Register Unit
LSU: Load Store Unit	**RS**: Reservation Station
IU: Integer Unit	**FPU**: Float Point Unit
BPU: Branch Processing Unit	

During the execution the states of the units are characterized by the instructions that are currently executed (eg. **IU**2 indicates that the integer unit executes the second instruction of the program).

5.2 Processor's modelization

Before starting the analysis we must provide a model of the processor. it is a simple program (see figure 3 which represents the fetch Unit) that describes the processor's behavior. The processor's model works as follow: every time the fetcher is free (instruction A), it tries to fetch an instruction from the instruction cache. First it associates an identifier to the instruction (instruction B), then the virtual address of the instruction is converted to a real one (instruction C), and it sends a request to the cache (instruction D). Finally it gets the time required to fetch the instruction (instruction E).
The instruction E shows that this description leads to build a time accurate model (TAM) (see figure 5). Indeed, every time the fetcher fetches an instruction, the program line E returns the time that this request takes. This behavior matches the definition of a time accurate model (Section 4). So each symbolic execution step of this model returns the next processor state $\mathcal{SC}'$ that is different from the current one as well

```
void Sequentialfetcher::fetch_instruction(bool*
        VirtualAddress ,int Identifier)
    {// test if the fetcher is free
     A. if (gettime()==0)
       { // associate each instruction to an identifier
      B. IdentifierVirtualAddress.insert( pair <int , bool
           *> (Identifier ,VirtualAddress));
       // convert the virtual address to a real one
      C. realaddress= pMemoryManagementUnit->
           EffectiveAddressToRealAddress(VirtualAddress);
       // fetch the instruction from the cache
      D. pControlcache->readrequest(VirtualAddress ,
           realaddress ,Identifier);
       // get the required time to fetch the instruction
      E. time= pControlcache->requesttime();
       }
    }
```

Figure 3: Processor's modelization: Fetch Unit (F)

as the number of clock cycles required to reach this system state. However, an executable timed model (figure 4) would split the instruction E into E1;E2....En steps (n depends on the scenario). Each step takes one clock cycle to be executed

5.3 Illustration of the conjoint symbolic execution

To illustrate the analysis method, we symbolically execute an assembly instruction (gray background) relying on the processor's model shown on figure 3. This is a load operation that

```
1.  lwz    %r1 ,off(@N
```

transfers a word from the memory to the register 1. Note that during the analysis each instruction has an identifier which represents its position in the program. Before explaining the execution, let us recall that we start the analysis with an empty pipeline ($\mathbf{P} = \mathbf{empty}$) and no information about the cache states ($\mathbf{IC} = \mathbf{DC} = \top$).

Starting from this initial state and relying on the processor's model, the analysis is carried out as follow:
On each clock cycle we test the fetcher (instruction A of the model), if it is busy we wait until the next clock cycle, else we associate an identifier to the instruction (instruction B). Then, the address of the instruction is converted from a virtual address to a real one (instruction C). After that, the fetcher sends a request to the instruction cache (instruction D). At this execution point we must first distinguish among two cases: the instruction is in the cache (cache hit) and the instruction is not in the cache (cache miss). We also must distinguish between the case where **the cache is idle**: we did not wait to request the instruction and the case where **the cache is busy**: we must first wait until the cache-line-reload operation terminates. So we can resume all the possible scenarios as follow:
(1) cache hit and the cache is idle (execution trace A;B;C;D;E).
(2) cache miss and the cache is idle (execution trace A;B;C;D;E1;E2..E5).
(3) cache hit and the cache is busy (execution trace A;B;C;D;E1;E2..E6).
(4) cache miss and the cache is busy (execution trace A;B;C;D;E1;E2..E12).
On the graphs presented by the figure 4 and 5 we see that,

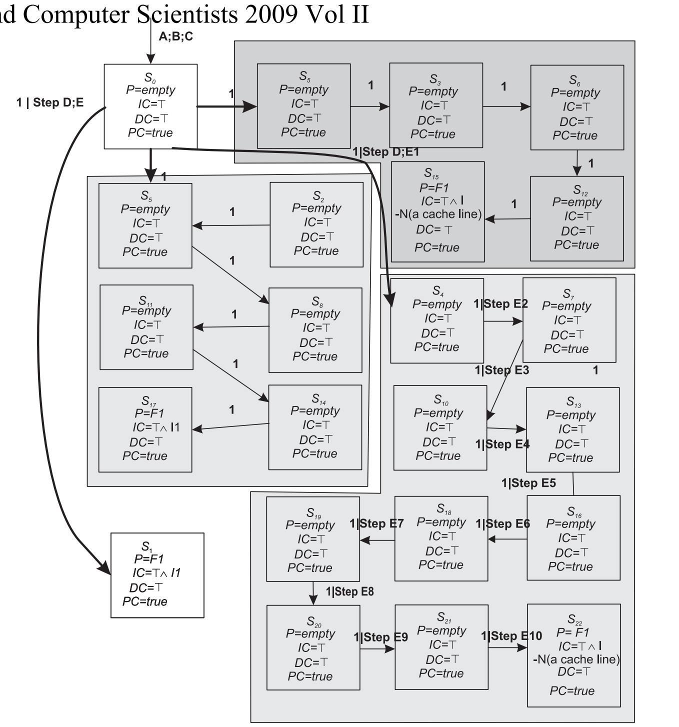

Figure 4: Symbolic execution steps of the executable timed model (ETM) for the assembly instruction

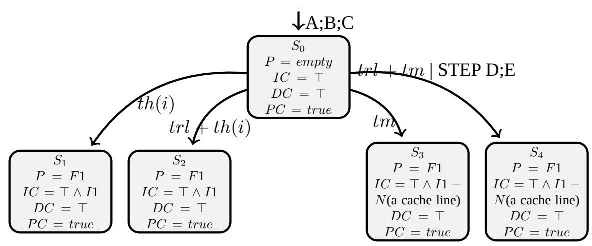

Figure 5: Symbolic execution steps of the time-accurate model (TAM) for the assembly instruction

concerning the scenario (1)–i.e., from the state S_0 to S_1, after one clock cycle: (A) the pipeline fetches the instruction thus its state moves from ($\mathbf{P} = \mathbf{empty}$) to ($\mathbf{P} = \mathbf{F1}$), (B) the instruction cache contains the instruction so its state moves from ($\mathbf{IC} = \top$) to ($\mathbf{IC} = \top \wedge I1$), (C) the cache state ($\mathbf{DC}$) as well as the PC stay at the same state. Now, in order to understand the interest behind abstracting the executable timed model, we focus on the last instruction of the model (instruction E). This instruction allows to know the time needed to get the instruction from the instruction cache. So in the presence of a cache miss (or pipeline stalls) if we choose to build the tree using an executable timed model (The simplest implementation which use the clock as the base cycle) it may lead to unnecessary intermediate states since the processor is waiting during the time returned by the instruction E. A more efficient implementation is the time-accurate model. It is achieved when returning the next processor state that is different from the current one as well as the number of clock cycles required to reach this processor state. Like we see on the figure 5 , this abstraction does not introduce any loss of precision. Thus we explore the same

paths ,to the scenarios presented above. But now instead of executing the program clock cycle by clock cycle, we compute immediately the set of pertinent "intermediate states". So the previous scenarios become:
(1) cache hit and the cache is idle takes $\mathbf{th}$ clock cycles.
(2) cache miss and the cache is idle takes $\mathbf{tm}$ clock cycles.
(3) cache hit and the cache is busy. takes $\mathbf{th}+\mathbf{trl}$ clock cycles.
(4) cache miss and the cache is busy takes $\mathbf{tm}+\mathbf{trl}$ clock cycles.
Now if we compare the graph on figure 5 with the one presented on figure 4 we conclude easily that:
- The number of the generated states has decreased (the execution trace has decreased from A;B;C;D;E0;E1;E2..En to A;B;C;D;E).
- Both of them contain the same information (we do not introduce any loss of precision).

6 Merging states

The symbolic execution allows to represent all the states that the processor may reach at each program point. So, the number of the generated states during the execution increases exponentially. Assuming that ρ represents the pipeline depth, σ denotes the maximal efficiency of the processor i.e. the number of instructions that are handled per clock-cycle, and η denotes the number of instructions of the code snippet, then an upper bound of the number of the states generated is: $3^{\sigma\ \eta\ \rho}$
To avoid this exponential states explosion, the second part of the analysis consists in merging those states relying on the loss of precision we are ready to accept. Indeed, we developed a merging policy (see [15] for details) to reduce the previous exponential increase to a linear one which is equal to:
$\gamma\ \sigma\ \eta\ 3^{\rho+\lambda}$ where γ is the maximum number of set of "intermediate states".

7 Conclusion and future work

We presented a part of our analysis method to compute an upper estimation of the WCET that can be parameterized by current complex hardware architecture, the only requirement is that an executable time-accurate model of the target system is available.
Instead of trying to build semi-automatically from the formal hardware description of the target system the black boxes of the analyzer that abstract the behavior of the target system [13], we have described a new approach that: Uses a simple model–.i.e, it is easy to develop this model or to modify it (1). Can easily adapted to complex hardware architectures (2). Takes into account all the dependencies and the interactions that happen between the processor's units during the execution (3).
This approach is currently being implemented and fully tested with time-accurate model of PPC 603e [3, 4] as well as multicore PPC 5554 processors.

References

[1] J. C. King.,"Symbolic Execution and Program Testing,", *Communications of the ACM*, V19, 7/76.

[2] J. A. Darringer., "The application of program verification techniques to hardware verification", *Annual ACM IEEE Design Automation Conference* pp. 376–381, 88

[3] MOTOROLA., *MPC603e EC603e RISC Microprocessors User's Manual with Supplement for PowerPC 603^{TM} Microprocessor*, 97

[4] Freescale Semiconductor., *PowerPC 603 RISC Microprocessors Technical Summary*, 94

[5] C. Ferdinand and D. Kastner and M. Langenbach and F. Martin and M. Schmidt and J. Schneider and H. Theiling and S. Thesing and R. Wilhelm., "Run-Time Guarantees for Real-Time Systems – The USES Approach", GI Jahrestagung, pp. 410–419, 99

[6] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios and R. Heckmann., "Computing the worst case execution time of an avionics program by abstract interpretation"

[7] P. Lokuciejewski, H. Falk, M. Schwarzer, P. Marwedel, H. Theiling., "Influence of procedure cloning on WCET prediction" *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign* Salzburg, Austria, pp. 137–142, 07

[8] R. Heckmann, C. Ferdinand., "Worst case execution time prediction by static program analysis" *18th Parallel and Distributed Processing Symposium*, pp. 125–134, 04/04

[9] P. Cousot., "Semantic Foundations of Program Analysis", *Program Flow Analysis: Theory and Applications*, New Jersey pp. 303–342, 81

[10] P. Cousot and R. Cousot., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", *Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, pp. 238–252, 77

[11] Y. Zhang., "Evaluation of Methods for Dynamic Time Analysis for CC-Systems AB" *Technical Report Mälardalen University*, 08/05

[12] D. B. Stewart., "Measuring Execution Time and Real-Time Performance" *Embedded Systems Conference*, San Francisco, 04/01

[13] M. Schlickling and M. Pister., "A Framework for Static Analysis of VHDL Code" *7th Intl. Workshop on WCET Analysis*, 07

[14] S.Thesing., *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*, 04

[15] B. Benhamamouch and B. Monsuez and F. Védrine., "Computing WCET using symbolic execution", *2nd International Workshop on Verification and Evaluation of Computer and Communication Systems. Leeds GB* , 08/08