Computing WCET using symbolic execution

Bilel Benhamamouch & Bruno Monsuez
UEI, ENSTA, 32 Bd Victor, 75739 Paris cedex 15, France
Bilel.Benhamamouch@ensta.fr, Bruno.Monsuez@ensta.fr

Franck Védérine
CEA, LIST, Boîte 65, Gif-sur-Yvette, F-91191 France
Franck.Vedrine@cea.fr

Keywords: WCET, real-time, symbolic execution, static analysis

Abstract

We propose a novel formal method to compute an upper estimation of the WCET that contains the loss of precision and also can be easily parametrized by the hardware architecture. Assuming that there exists an executable timed model of the hardware, we first use symbolic execution [5] to precisely infer the execution time for a given instruction flow. We secondly identify execution states that can be merged with no loss of precision. Depending on the loss of precision we are ready to accept, we finally merge execution paths that have similar execution times.

1. INTRODUCTION

We may define a real-time system as a system that must monitor and react immediately to the changing states of its environment. Design of real time systems is different from the design of other systems since temporal constraints are taken into account[9]. To simplify, we say that any system that is made up of several tasks that cooperate in order to build a set of functions prone to time constraints[9] (these constraints can be strict or not according to the nature of the application) is a real-time system.

The respect of these constraints is as significant as the corectness of the results. In other words real-time systems should not simply deliver correct results, they also must deliver them within the imposed times. The latter are dictated by the *Worst Cases Execution Times (WCET)*. Computing WCET are useful either to determine appropriate scheduling schemes for the tasks or to perform an overall schedulability analysis in order to guarantee that all timing constraints will be met.

With the increasing number and complexity of critical mission real time applications – in industry production for example, through control process (factories, nuclear thermal power stations) in transportation systems (satellites, planes, cars, trains) – precise estimation of WCET are required.

Computing program execution time has always been difficult. Dynamic methods as well as formal methods have received a lot of attention to allow precise estimation for the worst case execution time of code snippets [7]. However, current methods have some difficulties to cope with the increasing complexity of the hardware used to implement critical mission real time applications (super-scalar microprocessor, dual-core microprocessors).

In this paper, we present a novel formal method to compute an upper estimation of the WCET that contains the loss of precision and that can also be parametrized by current and future complex hardware architecture like super-scalar microprocessor, multi-processor systems.

The paper is organized as follows: we first introduce different frameworks used to compute WCET (Section 1) and we give an overview of our framework (Section 2). We then present the small

¹This work is a part of the APE project which is funded by the ANR

program (Section 3) as well as the significant architectural details of the hardware (Section 4) used to illustrate our approach. We then show how to compute the execution time of an instruction sequence using symbolic execution (Section 5) as well as how we can contain the explosion of the states generated by the symbolic execution (Section 6). Finally, we conclude and we present ongoing and future works (Section 7).

2. STATE OF THE ART

Various techniques were developed to achieve valuable estimation of the execution time. Those techniques can be classified in two families: the dynamic estimation (Section 2.1) and static analysis (Section 2.2) based techniques.

2.1. Dynamic estimation

Dynamic estimation consists in measuring the program execution times for some samples of pre-established input data. Therefore to determine the maximum execution time it would be necessary, either to be able to define for each program the adequate input data or to explore all the ways of its execution.

Dynamic methods estimate the WCET by measurements[12]. Methods to measure the execution time can be divided into three different categories: hardware (1) software (2) and hybrid methods (3). The hardware methods include measuring execution time using oscilloscopes, logic analyzers and emulators. The software methods are based on time-functions that for instance the operating systems provide. Hybrid methods combine the above two mentioned methods: small code snippets are inserted in the program that is being measured to trigger the hardware; when the program is executed, the code snippets will start or stop the hardware that monitors the target system and measures the execution time.

Many different methods exist to measure execution time[10], but there is no single best technique. Each technique is a compromise between multiple attributes, such as resolution, accuracy, granularity and implementation difficulty. The adequation of a given method will first depend on the hardware features and instrumentation tools available – some methods require special hardware features, others require a specific software application or measurement instrumentation to be available – and will also be impacted by the software design – execution time of programs that have multiple and inconsistent entry or exit points to the same piece of code are nearly impossible to measure[10].

More annoying is the fact that there are no guarantee that the longest execution time measured is the actual worst case execution time[12]. The WCET happens very rarely and the conditions to make it happen are normally unknown. So, all the dynamic methods can only measure one execution path at a time, and it is up to the user to find the inputs that will possibly cause the longest execution time. Practically, for standard architecture, the WCET is supposed to be less than the longest execution time found with an significant error margin – typically 30%.

From this, we can conclude that this method, although largely used in industry, misses exhaustiveness since the longest execution time measured may not be the worst.

2.2. Static analysis

Because of the inherent limitations of the dynamic estimation methods, formal methods have been developed to always find an upper-estimation of the WCET. Those methods work as follows, (I) loops and instructions branch are identified in order to analyze them separately from other instructions, (II) the remaining code is cut out in sequence and for each sequence the WCET is computed, (III) the results of the two previous stages is amalgamated to build a superset of all possible execution sequences and to identify the WCET.

With the appearance of new architectures intended to increase the computing performance of the processors (superscalar processor), computing the WCET for a code sequence requires to

abstract complex processor units like the cache memory, the pipeline, the speculative execution or re-scheduling of instructions. Among all the various approaches, the one developed by R. Wilhelm and the AbsInt team[3] is certainly the most mature one.

Figure 1: AbsInt's WCET estimation method

As represented in figure 1, WCET computation is carried out by a succession of analyses[3, 7]. First the control flow graph (CFG) is extracted from the binary code. This CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The nodes of the graph are blocks of code, called basic blocks, while the edges show how the execution of the program passes from one block to another one. A value analysis is carried out on this CFG to determine the memory areas that may be reached during program execution. The result of this analysis is exploited by the cache analysis that determine potential cache misses and certain cache hits. Then a pipeline analysis computes for each execution point of the analyzed program the possible states of the pipeline. Finally, the results obtained during the preceding stages, are finally exploited jointly with the source code by a last analysis called path analysis. This analysis is based on linear programming techniques, what enables it to produce the longest execution path.

Value analysis, cache analysis and pipeline analysis are all based on abstract interpretation that abstract concrete values to abstract values (a whole of concrete values could thus be represented by an abstract value). Each black box provide an abstract semantics of the hardware that describe the behavior of those components on the abstract values.

The AbsInt approach is represented by a flow of black boxes[3, 7] that abstract the behavior of hardware components. 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 abstract the hardware semantics.

Formal methods have three main drawbacks; (1) those methods explore a superset of all execution paths so that WCETs for unfeasible execution paths are taken into account; (2) to avoid the state explosion execution paths are merged, that may also conducts to an over-exaggerated approximation of the execution time; (3) the analyser must explicitly support the target platform and must provide valuable abstraction of the hardware components that compose the target platform.

3. AN EXTENSION TO THE FORMAL METHODS: CUSTOMIZING THE WCET COMPUTATION BY AN EXECUTABLE TIMED-MODEL OF THE TARGET SYSTEM

To mitigate the drawbacks cited in the previous section (Section 2.2), we propose a new approach that extends the classical framework for computing the worst-case execution of a sequence of code with no loops or branch instruction (phase II of the workflow of formal methods). 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 give 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*.

Conjoint symbolic execution of program code and executable model of the processor: 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. For instance, after each cache miss the PowerPC 603 initiates a memory transaction that loads a cache line (4 double words). If during execution a cache miss occurs when accessing a double word, the cache gets updated and accessing the double words that follows immediately the loaded double word will result in a cache hit.

Backward execution paths merging with symbolic execution lookup policy To avoid state explosion that is inherent to formal methods, similar states must be merged. However careless states merging policies may conduct to very large over estimation of the WCET. Since the proposed framework does not impose any requirement on the hardware semantics, we cannot define a static merging policy as it is the case in the AbsInt framework but we must dynamically measure the impact of merging states on the WCET computation before deciding to merge the states. Our policy works as follow: we first browse backward the execution paths to identify states that may be merged (backward execution) then we suppose we merge the states and we measure how the execution of the next instruction is impacted (symbolic execution lookup). Depending on the loss of precision we are ready to accept, a final decision on merging the execution paths can be made.

In the next section, we present the program and the target processor (PowerPC 603) that we use to illustrate how the method works. In section 5 we describe more deeply the symbolic execution. In section 6, we finally present the algorithm that implements the merging policy.

Figure 2: Our WCET estimation method

4. A SMALL EXAMPLE TO ILLUSTRATE HOW WCET ARE COMPUTED

To illustrate the technique of computing WCET using symbolic execution, we will apply this technique to a short sequence of code running on a typical embedded processor, the PPC 603.

4.1. The code sequence

We use the code snippet in figure 3 as a running example. Figure 4 shows the sequence PPC instructions that implement the C function `acquisition`.

```

int N,i,j;
int measure_table[N];
...
void acquisition(int measure)
{
    if (measure > measure_table[N-1])
        measure_table[N-1]=measure;
    else if (measure < measure_table[0])
        measure_table[0]=measure;
}

```

Figure 3: Running example: the function `acquisition` is triggered with each clock signal and refreshes the first or the last element of the array `measure_table`

We identify each instruction of the binary code by a unique identifier – in the present case, the position of the instruction in the code sequence.

<pre> BEGIN 1 lwz %r1, off (PTR_N) 2 lwz %r2, off (PTR_measure) 3 mulli %r1,%r1,4 4 addi %r1,%r1,-4 5 addi %r1,%r1, off 6 lwz %r3,%r1 (PTR_MeasureTable) 7 cmp %r2,%r3 8 ble \$LN1 9 lwz %r4, off (PTR_N) 10 lwz %r5, off (PTR_measure) </pre>	<pre> 11 mulli %r4,%r4,4 12 addi %r4,%r4,-4 13 addi %r4,%r4, off 14 stw %r5, %r4 (PTR_MeasureTable) 15 b \$EXIT \$LN1: 16 lwz %r6, off (PTR_MeasureTable) 17 cmp %r2,%r6 18 bge \$EXIT 19 stw %r2, off (PTR_MeasureTable) \$EXIT: 20 END </pre>
---	--

Figure 4: The PowerPC binary code of the function `acquisition`

4.2. The target processor

The simple code sequence will be executed on a PPC 603, a micro-processor typically used on embedded applications. We must provide an executable timed model of the hardware used during our analysis [4]. This last, can be formulated in different hardware description languages like SystemC, VHDL or Verilog but also C++.

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 A processor unit state $SC[u]$ is a minimal set of properties that allow to define what is the next operation that this unit u will perform.

Definition 2 The state of the target system S is the product of all the states of the units that compose this system S : $S = (\prod_{u \text{ unit of } S} SC[u])$.

The model must be time-accurate, that means that it must preserve the time (number of clock cycles) the processor needs to compute an instruction. The simplest implementation of time-accurate model use the clock as the base cycle (clock accuracy). So for each clock cycle, it computes the new state of the processor. However, in the presence of cache miss and pipeline stall, it may lead to unnecessary intermediate states, since the processor is waiting for some data. A more efficient implementation of time-accurate model 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 3 An executable clock-accurate model is an executable function that maps a processor state $s \in S$ to next processor state s at the next clock cycle.

Definition 4 *An executable time-accurate model is an executable function that maps a processor state $s \in S$ to the pair of a processor state $s \in S$ and the time $t \in T$ needed to reach this processor state.*

Specific implementation details for the PPC 603 The PowerPC 603 is a low-power super-scalar implementation of the PowerPC processor family. It provides independent on-chip, 8-Kbyte, two-way set-associative, physically addressed caches for instructions and data. Instructions can execute out of order for increased performance; We give some pertinent details about the instruction pipeline in the PPC 603, for more details see[6, 4].

This processor integrates 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). To those units we must also add the pipeline stages:

Fetch At most two instructions can be retrieved simultaneously from the memory system and the location of the next instructions fetch is computed. Branch instruction can be decoded by the BPU during the fetch stage and fold out before the dispatch stage if possible.

Dispatch This stage decodes the instructions supplied by the instruction fetch stage and determines which of the instructions are eligible to be dispatched in the current cycle. In addition, the source operands of the instructions are read from the appropriate register file and dispatched with the instruction to the execute pipeline stage. At the end of the dispatch pipeline stage, the dispatched instructions and their operands are latched by the appropriate execution unit.

Execute Each execution unit that has an executable instruction executes the selected instruction (perhaps over multiple cycles), writes the instruction's result into the appropriate rename register, and notifies the completion stage that the instruction has finished execution. In the case of an internal exception, the execution unit reports the exception to the completion/writeback pipeline stage and discontinues instruction execution until the exception is handled. Execution of most floating-point instructions is pipelined within the FPU allowing up to three instructions to be executing in the FPU concurrently. The pipeline stages for the floating-point unit are multiply, add, and round-convert. Execution of most load/store instructions is also pipelined. The load/store unit has two pipeline stages. The first stage is for effective address calculation and MMU translation and the second stage is for accessing the data in the cache. During the execution we represent only the execution units that are busy.

Complete/writeback pipeline stage maintains the correct architectural machine state and transfers the contents of the rename registers to the GPRs and FPRs as instructions are retired. If the completion logic detects an instruction causing an exception, all following instructions are canceled, their execution results in rename registers are discarded, and instructions are fetched from the correct instruction stream.

The following table summarizes all the processor units that compose the PPC 603 as well as the atomic times that are associated to a particular cache operation.

Processor units	Times taken by a cache operation
F: Fetcher	th: time associated to a cache hit.
D: Dispatcher	tm: time associated to a cache miss.
BPU: Branch Processing Unit	trl: time associated to a cache line reloading.
LSU: Load Store Unit	(d): data.
IU: Integer Unit	(i): instruction.
FPU: Float Point Unit	
SRU: System Register Unit	
CU: Completion Unit	
RS: reservation station	

States of each unit and pipeline stage are abstracted by the instructions that are currently executed. For example, *IU1* indicates that the integer unit executes the first instruction of the program. The state of the units and pipeline stages of processor evolves as follows:

Figure 5: The evolution of system states

To the states of the pipeline as well as the computation unit, we must add the state of the data as well as the instruction cache. The state of the data and instruction cache must be able to characterize if a data is present or is not in the data or in the instruction cache.

5. CONJOINT SYMBOLIC EXECUTION OF BINARY CODE AND TIME-ACCURATE SYSTEM MODEL

5.1. Background: Symbolic execution

The main idea behind symbolic execution [5, 2] is to use symbolic values, instead of actual data to represent the values of program variables as well as the input values. As a result, the output values computed by a program are expressed as a function of symbolic value. Evaluation of assignments is done naturally, the left-hand sided 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 :

- $PC \supset R$ and $PC \not\supset \neg R$: In this first case, the expression is always true, the execution continues with the conditional code sequence.
- $PC \supset \neg R$ and $PC \not\supset R$: In this first case, 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 $PC_{\text{true}} = PC \wedge R$ and $PC_{\text{false}} = PC \wedge \neg R$ and we continue the concurrent execution of the condition code sequence with PC_{true} and the "else" code sequence or the code located after the conditional code sequence with the path condition PC_{false} .

The state of a symbolically executed binary program includes the system state as defined in the previous section ST , the execution time ET , the symbolic values SV , the path condition PC as well as the next instruction to be executed NI .

5.2. Conjoint symbolic execution

The time-accurate model is symbolically executed for the given code snippet. As described before, the time-accurate model takes as an input a current state S of the system and returns the next processor state 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 state S of the system and returns a set of final states $\{S_1, \dots, S_n\}$ returned by the time-accurate model.

Definition 5 If $S = [NI, ET, ST, SV, PC]$ is a valid system state, we call “intermediate states” the states $\{S_1, \dots, S_n\}$ generated by one step of symbolic execution of the time-accurate model when starting the execution with the state S .

We call a “symbolic step” a transition that maps the state S to one of the “intermediate states” S_i .

The algorithm that symbolically executes the code snippet works as follows:

- **Initialization:** Adds to the set of states to be evaluated the initial instruction.
 $states = \{ [NI = \text{first binary instruction}, ET = 0, PC = \{ \text{true} \}]$
 $ST = \{ \text{the pipeline is empty, the state of the instruction cache and}$
 $\text{data cache as well as the data are unknown} \}$
- **Propagation:**
 while($states$ is not empty)
 - removes state $S = [NI, ET, ST, SV, PC]$ from the $states$
 - computes the “intermediate states” $\{S_1, \dots, S_n\}$ by symbolically executing the time-accurate model starting the execution with the state S .
 - adds all the states of $\{S_1, \dots, S_n\}$ that are not the final state generated by the last instruction of the code sequence to $states$
 - adds all the states of $\{S_1, \dots, S_n\}$ that are final states generated by the last instruction of the code sequence to the set of final states $final_states$
- **Termination:** Returns the worst-case time of the computed time in the set of final states $final_states$.

5.3. Illustration of the symbolic execution

We choose to begin the execution with an empty pipeline ($P = \text{empty}$) and unknown caches states ($IC = DC = \top$). Concerning the pipeline initial state, this assumption was motivated by the fact that beginning an execution with an unknown pipeline state implies having some information about the running application, which implies other assumptions.

So beginning from this initial state the execution continues as follow:

Figure 6: Crossing of the program and the processor description

According to the processor description [4], two instructions can be simultaneously fetched from the instruction cache. But when the Fetcher attempts to fetch instructions from the instruction cache, the cache may or may not be able to immediately respond to the request. There are two

scenarios that may be encountered by the Fetcher. The first scenario is when the cache is idle. In this case, the cache responds with the requested instructions on the next clock cycle, if obviously the instructions requested are in the cache (cache hit), otherwise a memory transaction is required to bring the instruction into the cache (cache miss).

The second scenario occurs if at the time the Fetcher requests instructions, the cache is busy due to a cache-line-reload operation. When this case arises, the cache will be inaccessible until the reload operation is complete.

The figure 6 gives a small overview of the evolution of the graph generated according to our use of symbolic execution. The chosen notation lets know precisely the state in which the processor is at certain time and how much time it will take for immediate changes.

6. MERGING STATES: FROM EXPONENTIAL TOWARDS LINEAR COMPLEXITY

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, assuming that σ denotes the average efficiency of the processor – the number of instructions that are handled per clock-cycle – and assuming that η denotes the number of instructions of the code snippet, an upper bound of the number of the state generated is: $4^{\sigma \eta \rho}$

To avoid this exponential explosion of states, states must be merged. However since the target system is described by an executable model, the merging policy cannot be defined statically, using for example widening operators [11]. We must take the decision to merge or not merge the state depending on the result and the evolution of the symbolic evaluation. Therefore we propose to develop a merging policy to reduce this exponential increase in a linear one. The idea behind the proposed merging method is:

- starting with an instruction, we build the execution tree that starts with this instruction and each path of this execution tree has at least a number of symbolic steps that is equal to the number of pipeline stages.
- we then first identify the equivalent states in the execution tree and merge those equivalent states. Those states can be merged with no loss of precision.
- starting with the merged states, we first browse backward the execution paths to identify states that may be merged (backward execution) with some loss of precision and measures how the time estimated may be impacted in the next future. To estimate the impact of the merging the states $S_1, \dots, S_i$, we first build for each state S_i that may be merged the symbolic instruction tree that starts with these state S_i . Then we merge those states $S_{\text{merged}} = S_1 \sqcup \dots \sqcup S_i$, and build the symbolic instruction tree that starts with the state S_i . for a given number of symbolic steps (symbolic execution lookup), we build the symbolic instruction tree that starts with the merged state S_{merged} and we compare the difference of the estimated execution time obtained when merging the states and the execution time obtained when the states are not merged.
- Depending on the loss of precision we are ready to accept, we merge all the states that can be merged so that the number of “intermediate states” is less than a fixed upper-bound γ .

With respect to this policy, the complexity is bounded by the following formula: $\gamma \sigma \eta 4^{\rho+\lambda}$ where:

- γ is the maximum number of set of “intermediate states”.
- λ denotes how many “symbolic steps” must be achieved to measure the impact of merging similar states.

6.1. Revisiting the algorithm

We enhance the algorithm that performs the symbolic evaluation with the merging policy we have defined in the previous section. To implement the algorithm, we must provide two equivalence relations: strong and weak similitude.

We say that two states S_1 and S_2 are strongly similar if except the path conditions PC_1 and PC_2 and the estimated time ET_1 and ET_2 that may be different but all other component states are equivalents.

We say that two states S_1 and S_2 are weakly similar if except the path conditions PC_1 and PC_2 and the estimated time ET_1 and ET_2 that may be different, the difference for all other states components are small. The definition of the what is "small" must be provided either by a formal distance between states or a specific heuristic.

// Initialization:

Adds to the set of states to be evaluated the initial instruction.

$states \leftarrow \{ [NI = \text{first binary instruction}, ET = 0, PC = \{ \text{true} \}], ST = \{ \text{the pipeline is empty, the state of the instruction cache and data cache as well as the data are unknown} \}$

// Propagation:

while $states$ is not empty **do**

Removes state $S = [NI, ET, ST, SV, PC]$ from the $states$ and computes the symbolic evaluation tree $ET(S)$ starting with the state $states$ for at least ρ "symbolic steps".

Computes *equivalence classes* $EQ = \{e_1, \dots, e_k\}$ of $ET(S)$ w.r.t the defined **strong similitude relation**.

foreach state $e = \{e_1, \dots, e_k\}$ in EQ **do**

// Starts the forward analysis

$S_\delta \leftarrow \bigsqcup_{i=1 \dots k} e_i$

Starting with the states e_1 to e_k , merges all the successor of the states that are strongly similar and removes the merged states from all the equivalence classes in e .

// Starts the backward analysis

$backward_paths \leftarrow \{[e_1], \dots, [e_k], \}$, $continue \leftarrow \text{true}$

while $continue$ **do**

Gets the set of the precessing states S_b^i of S_δ .

if All the precessing states S_b^i are weakly similar **then**

Computes the symbolic evaluation tree $ET(S_\delta)$ where $S_\delta = \bigsqcup_{i=1 \dots k} S_b^i$.

Estimates the error between the time computed after the states have been merged and before the states have been merged.

if If the loss of precision is acceptable **then**

Merges all the states S_b^i and removes the merged states from all the equivalent classes in e .

$S_\delta \leftarrow \bigsqcup_{i=1 \dots k} S_b^i$.

else

$continue \leftarrow \text{false}$

else

$continue \leftarrow \text{false}$

adds all the final states of $ET(S)$ that are not the final states generated by the last instruction of the code sequence to $states$ and all the final states of $ET(S)$ that are final states generated by the last instruction of the code sequence to $final_states$

// Termination:

Returns the worst-case time of the computed time in the set of final states $final_states$.

6.2. Revisiting the example

Now we apply the algorithm presented above (Section 6.1) to the example introduced in Section 4.1.

The two paths shown on figure 7 are generated during the symbolic execution of the code sequence of the function *acquisition*. During the symbolic execution many paths converge towards a strong similar state. We qualify this state as *discriminant* one if: (1) It is an execution point which brings together paths which have different historical execution, (2) After this common state, all the paths that converge towards it, will have the same or approximately the same behavior.

Instead of trying to build semi-automatically from the formal hardware description of the target system the black boxes of the analyser that abstract the behavior of the target system [8], we have described a new approach that use the formal hardware description as a formal description that we can symbolically execute. This approach is currently being implemented and will be fully tested with time-accurate model of PPC 603e as well as PPC 5554 processors.

As we have seen in the last section, the quality of the results heavily depends on the ability to merge generated states with very little loss of precision. We have proposed an original policy to estimate the potential loss of precision that merging two or more states may induce. This policy is novel in at least two points: (1) the forward analysis only build the set of reachable states, it is during the backward analysis that decisions are made about merging states, (2) before merging the states, a forward analysis is performed to estimate the loss of precision on the next instruction that may be the consequence of the merge.

However, we still need to provide two relation classes that define the strong similitude and the weak similitude of two states. We already have developped some heuristics that seem to works but we certainly need more work to better understand the properties that those heuristics should preserve and how those relation classes impact the final results.

Bibliography

- [1] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 142–151, New York, NY, USA, 2001. ACM.
- [2] John A Darringer. The application of program verification techniques to hardware verification. *Annual ACM IEEE Design Automation Conference*, pages 376–381, 1988.
- [3] Christian Ferdinand, Daniel Kastner, Marc Langenbach, Florian Martin, Michael Schmidt, Jorn Schneider, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Run-time guarantees for real-time systems – the USES approach. In *GI Jahrestagung*, pages 410–419, 1999.
- [4] Freescale Semiconductor, Inc. *PowerPC 603TM RISC Microprocessors Technical Summary*, 1994.
- [5] James C. King. Symbolic execution and program testing. *Communications of the ACM (Association for Computing Machinery)*, 19(7), July 1976.
- [6] MOTOROLA. *MPC603e EC603eTM RISC Microprocessors User's Manual with Supplement for PowerPC 603TM Microprocessor*, 1997.
- [7] Christian Ferdinand Reinhold Heckmann. Worst case execution time prediction by static program analysis. *Parallel and Distributed Processing Symposium. Proceedings. 18th International Volume*, pages 125–134, April 2004.
- [8] Marc Schlickling and Markus Pister. A framework for static analysis of vhdl code. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- [9] J. Stankovic. Distributed real-time computing: The next generation. *Special issue of Journal of the Society of Instrument and Control Engineers of Japan*, 31(7):726–736, 1992.
- [10] David B. Stewart. Measuring execution time and real-time performance. *Embedded Systems Conference, San Francisco, CA.*, April 2001.
- [11] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, November 2004.
- [12] Yina Zhang. Evaluation of methods for dynamic time analysis for cc-systems ab. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-182/2005-1-SE, Mälardalen University, August 2005.