Improving Processor Hardware Compiled Cycle Accurate Simulation using Program Abstraction

Bullich Adrien
IRCCyN
1 rue de la Noë
Nantes, France
Adrien.Bullich@irccyn.ec-nantes.fr

Briday Mikaël
IRCCyN
1 rue de la Noë
Nantes, France
Mikael.Briday@irccyn.ec-nantes.fr

Béchennec Jean-Luc
IRCCyN
1 rue de la Noë
Nantes, France
Jean-Luc.Bechennec@irccyn.ec-nantes.fr

Trinquet Yvon
IRCCyN
1 rue de la Noë
Nantes, France
Yvon.Trinquet@irccyn.ec-nantes.fr

ABSTRACT

Verification is an important step in the development of real-time embedded systems. The validation of a real-time system uses a timing accurate simulator and, when the actual binary code is used, a cycle accurate simulator (CAS). However, a CAS is slow especially when the simulated processor is complex and the application is big. One way to improve the speed of a CAS is to use compiled simulation. In this scheme, the application binary code model is merged with the processor model. This allows to remove operations from the simulator and to speed up it. In this paper, we show how to use an abstraction of the program and improve the handling of functions calls. The resulted simulator is temporally and functionally equivalent. This technique improves simulation speed by more than 50% over the speed of an interpreted CAS[1].

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance Analysis and Design Aids

General Terms

Verification, Performance, Algorithms

Keywords

Processor Hardware Simulation; Compiled Simulation; Cycle Accurate Simulation; Real-Time Systems

[1]This work is supported by ANR project ImpRo ANR-2010-BLAN-0317

1. INTRODUCTION

Verification and validation of real-time embedded systems is an important part of their development because a failure can lead to dramatic consequences. In addition to a correct functional behavior, they also have to respect timing constraints.

In the last steps of the validation process, when the actual binary code of the application is available and when the actual hardware is known, formal methods become unsuitable because the formal model of the system includes a lot of details. As a result, it is huge and leads to combinatorial explosion. Instead validation by using precise models of the hardware, *e.g.* a CAS simulator is greatly adapted at this stage of the validation process.

The choice of simulation tools depends on the field studied and the required objectives. They determine the abstraction level required by the simulator. Lower is this abstraction level, higher is the complexity of the simulator. The handwritten development of these simulators is a long and error prone work. To facilitate the generation of a simulator, a hardware Architecture Description Language (ADL) [12] can be used. The processor is described using a dedicated Domain Specific Language, and an associated compiler is provided to generate a simulator. The work presented in this paper is part of the HARMLESS project [10]. HARMLESS is a language too [11] which allows the generation of both a functional simulator, *i.e.* Instruction Set Simulator (ISS), and a CAS. The later gives a temporal information in addition to the functional behavior, but at the cost of a significant computation time.

A Cycle Accurate Simulator should take into account micro-architecture parts of the processor that have an impact on timings. This particularly implies the modeling of the pipeline which leads to a greater complexity of the simulator. Moreover, it has a huge simulation performance cost compared to an ISS. The computation time required for simulation is, however, a real handicap during the validation process, in particular when running a large amount of different scenarios.

In this paper, we focus on techniques to reduce the execution time of a CAS, using the compiled simulation. We propose a model allowing to take into account efficiently most

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254792

of the applications. Then, we improve the compiled simulation to further reduce the execution time by abstracting the application.

The paper is organized as follows: Section 2 presents related works; Section 3 introduces the basis of interpreted simulation; Section 4 develops a model of compiled simulation, that takes into account the software floating point computation problem; Section 5 describes the application abstraction done to further speed up the simulation; Section 6 presents results; finally, Section 7 summarizes our different contributions.

2. RELATED WORKS

Flexible simulator generation from a processor model supposes the use of a hardware ADL. We can find many hardware ADLs in the literature. Hardware ADLs may focus on the functional aspects only. In this case, a description of the instruction set is provided. It allows to generate an ISS. For example, nML [4] and ISDL [6] are this kind of hardware ADLs. Structural hardware ADLs add the ability to do a micro-architecture description, in order to simulate the temporal behavior. LISA [13], MADL [14] and HARMLESS [9, 11] can generate both an ISS and a CAS.

We can also find in the literature for ISS the difference between interpreted simulators and compiled simulators. An interpreted simulator, for each binary instruction, does the following steps: instruction fetch, instruction decode, and instruction execution. That is the same steps than the hardware simulated. A CAS is usually implemented as an interpreted simulator. In addition to an ISS, it computes instructions dependencies, controls concurrent accesses to the buses, register files, and generally any computing resource of the architecture.

A compiled simulator is attached to a particular program. Since the binary executable is known during the compilation stage, it is possible to move from the execution stage to the compilation one all the tasks that depend on the executed instruction only. This move leads to a shorter execution time and the simulator exhibits better performance than the interpreted one. However, the compilation time is longer. However, if the execution is done more times than the compilation, classically several execution are done for one compilation, we can get a global gain of time. The main problem remains. The compiled simulator is less flexible that the interpreted one, because the latter is not attached to a particular program: if one needs to simulate another program, another compilation must be performed.

The interpreted simulation is implemented for ISS and CAS. But the compiled simulation is mainly used by ISS. It consists in Binary Translation (BT) ([3] or [1]). The principle of BT is to translate the binary executable we want to simulate to a native binary of the host simulation platform. Then, this native binary could be executed directly on the host simulation platform.

Some methods exist to implement compiled simulation for CAS. The technique of BT cannot easily be adapted to CAS, but it is not impossible: [8] couples interpreted parts and translated parts. Statistical approaches are another example and Cycle Approximate Simulators are based on the sampling of instructions [15]. But they are not exactly equivalent to a CAS, because of errors margin.

The use of compiled simulation for CAS implies many restrictions: the static determination of the evolution of the micro-architecture is difficult. That is the reason why the technique is so few employed to speed up CAS. However, in [2] a model has been proposed. It allows to get a gain of the execution time. In this paper, we propose to improve this gain and to dismiss one major restriction: software floating point computation.

3. INTERPRETED SIMULATION MODEL

To assess the performance of the compiled simulation, we need a point of comparison: the associated interpreted approach. We present, in this section, the interpreted model of the Cycle Accurate Simulator based on HARMLESS [10]. It is also the base of our compiled model.

The procedure that the interpreted simulator follows is to decode instructions of the application code and to execute them. Our system includes the instruction set, the memory model and all the micro-architecture related parts that alter timings. The Figure 1 presents the development chain used by the interpreted simulation.

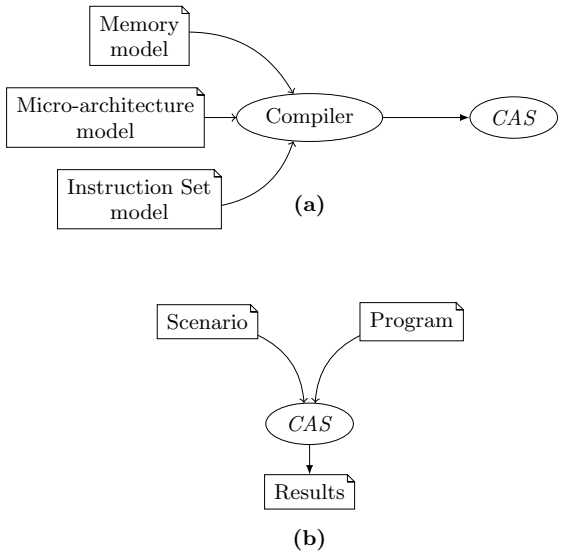

Figure 1: The development of a CAS requires the modeling of the instruction set, the memory and the micro-architecture. (a): compilation, (b): execution

The main micro-architectural feature which is the costliest to simulate and that alters timings is the processor pipeline. It allows to execute instructions with some parallelism. Ideally, each instruction in each pipeline stage progresses to the next stage each cycle.

However, *hazards* can block instructions in pipeline stages. Hazards are classified into three categories [7]:

- *structural hazards* result from a lack of hardware resources;
- *data hazards* are caused by data dependencies between instructions (for example between stages W and D in Figure 2);
- and *control hazards*, are caused by branching policy. When a branch is taken, instructions that just follow the branch must be flushed, according to the branch delay.

When a hazard is encountered, it is solved by introducing a *pipeline stall*: a part or all parts of the pipeline is stopped.

In this paper, we only consider sequential pipelines, *i.e.*, there are neither pipelines working in parallel, nor forking pipelines. We model the pipeline behavior by using an automaton. A state represents the pipeline state at a particular time, as we can see in the Figure 2.

The system can be modeled by a discrete transition system, because a transition is taken at each cycle (see Figure 2) [10].

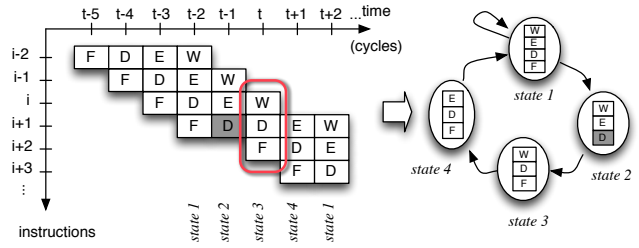

Figure 2: A state of the automaton represents the state of the pipeline at a given cycle. Here, the pipeline has 4 stages. F: instruction fetch, D: instruction decode and registers read, E: instruction execution, and W: result write into a register.

Then the definition of the interpreted model is the following. A state represents the system in a particular cycle, and is defined by:

- which instruction is in each stage of the pipeline;
- the state of internal resources.

Internal resources are elements of the micro-architecture, affecting the pipeline, and used only by it. They affect the pipeline by their availability. If the internal resource is available, the progression of an instruction in the pipeline is allowed, else it is not allowed. We consider stages of the pipeline as internal resources.

In order to simplify the implementation of the model, we group together instructions to form *instruction classes*, when they use the same resources in the same pipeline stage. For instance, arithmetic instructions that read registers, make a calculation and write the result into a third register form an instruction class. At run time, no internal resource is needed, because they depend only on the instruction class and on the pipeline stage.

The simulation performs from a state to another by firing a transition. A transition represents a discrete event of one cycle time, and is determined by the state of *external resources* and the next instruction class that enters the first stage of the pipeline.

External resources are quite similar to *internal resources*, but they are not solely used by the pipeline. Their state is defined in other micro-architecture parts, such as a memory caches. These external resources have an influence on the evolution of instructions in the pipeline. Since their state are only determined dynamically in function of the other architectural parts, their availability is determined during the execution.

We abstract the content of states. The simulation requires information from the execution of the automaton: it is gathered on transitions, with a label. This information is a set of *notifications*, which signal if a particular event happens or not.

Now, we can formalize the interpreted model. Let IA be an automaton defined by $\{S, s_0, ER, IC, N, T\}$, where:

- S is the set of states;
- s_0 is the initial state (empty pipeline) in S;
- ER is the first alphabet of actions (external resources);
- IC is the second alphabet of actions (instruction classes);
- N is the alphabet of labels (notifications);
- T is the transition function in $S \times ER \times IC \times N \times S$.

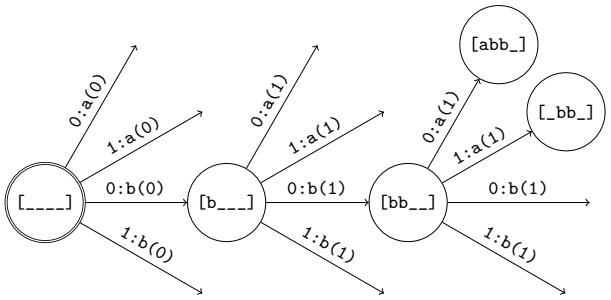

Figure 3: Automaton in interpreted simulation: a transition `0:b(1)` means that the external resource is available `0:b(1)` , the instruction `0:b(1)` may enter the pipeline and the notification happens `0:b(1)`

In the Figure 3, an example is presented. The notation `[bb__]` represents the state of the 4-stages pipeline: it means that an instruction of class `b` is in the first stage and that another instruction of class `b` is in the second stage. Other stages are empty. This simple example has only two instruction classes: `a` and `b`. We have only one notification that represents the *entry of an instruction in the second stage of the pipeline*. There is only one external resource. The instruction class `a` needs to take the external resource to enter the pipeline.

The execution of the simulation proceeds by exploring the automaton. The conditions to determine the next state of the automaton are the state of the external resources and the instruction class that will enter the pipeline. The notifications labeled on the taken transition give to the simulation engine information to interact with other micro-architectural parts.

4. COMCAS MODEL WITH PUSHDOWN AUTOMATON

In this section, we broach the problem of the adaptation of the interpreted model into a compiled one.

The compiled simulation is opposed to the interpreted simulation in the repartition of tasks between compilation step and execution step. The main goal of the compiled simulation is to move the analysis of the program from the execution step to the compilation step. This is particularly interesting as the compilation step is done only once. On one hand, the compilation step is slower with the compiled

simulation than with the interpreted simulation, because the program is analyzed and the simulation engine is optimized for this particular program. On the other hand, the execution step is much faster. This approach is less flexible in the debug sequence of an application, when the code of the application is often updated. However, it is particularly interesting in the test sequence of an application, when many scenarios have to be evaluated on the same binary code.

Figure 4 shows the development chain for compiled simulation. It has to be compared to figure 1. In the compiled simulation, the program is at the beginning of the development chain and is part of the build step of the simulator.

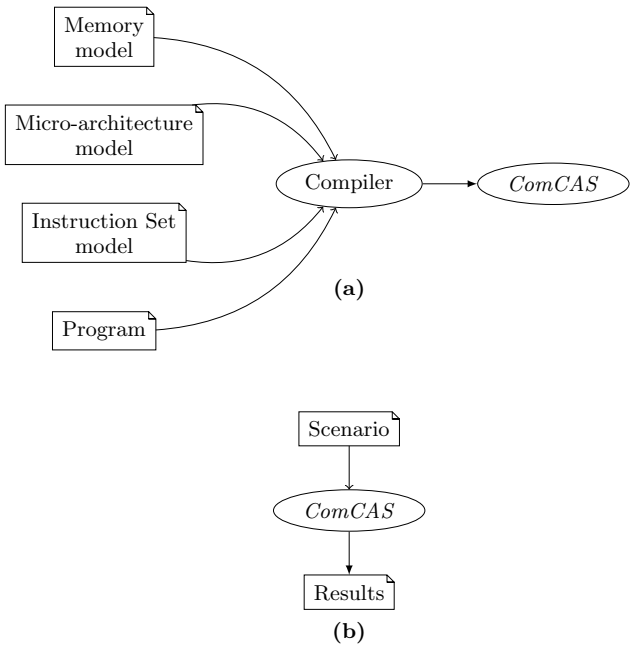

Figure 4: The development of a compiled CAS requires to move the program analysis at the compilation step. (a): compilation, (b): execution

A model of compiled simulation, temporally and functionally equivalent with the interpreted one, has been proposed in [2] for HARMLESS and is the basis of our contribution. The analysis of the program requires information about the program: its semantics, the Program Counter (PC) and the stack of function calls. Compared to interpreted simulation, the model is extended to include the Program Counter (PC) and the stack of function calls.

In addition, the control of data dependencies has been moved to the compilation phase. This move reduced by 45% the execution time, in average. Since the compiled simulation analyzes statically the program, it can not take into account any indirect branches when the target PC is unkown at compilation phase.

The management of functions in the program uses a particular indirect branch: the function return. The target of this branch is determined by a function call stack. When the function is called, the following PC is pushed onto the stack. Then, when a function return is read, the target is at the top of the stack. This is to simulate this particular indirect branch that the stack of function calls is included in the model. Because, we know the state of the stack in each state of the automaton, we can determine statically the target of function returns. But, this technique has a drawback: the size of the model is increased. Indeed, the technique is equivalent with the duplication of states each time a function is called. The more a function is called, the bigger is the automaton. For example, if floating point computations are managed with functions, the size of the automaton increases considerably, and the generation of the simulator becomes unfeasible.

In this paper, we propose an improved model that takes into account function returns, without penalty on the size of the automaton. This improvement allows to simulate the programs using software floating point computation. Moreover, it becomes possible to simulate recursive programs.

If the management of function call stack is not processed statically, then it must be processed dynamically. The execution of a program is a context-free language, and could be described by a Pushdown Automaton (PA). Thanks to this model, it is possible to describe our system without including function call stack. In our contribution, our system is only defined by:

- the state of the pipeline: which instruction is in each stage;
- the state of internal resources;
- and the position in the program (the Program Counter).

The system changes its state each cycle according to the availability of external resources and to the top element of the stack of the PA.

Let PA be a Pushdown Automaton defined by $\{S, s_0, ER, N, Z, z_0, T\}$, where:

- S is the set of states;
- s_0 is the initial state in S;
- ER is the input alphabet (external resources);
- N is the label alphabet (notifications);
- Z is the stack alphabet;
- z_0 is the symbol for the bottom of the stack, in Z;
- T is the transition function in $S \times ER \times (Z \cup \epsilon) \times S \times N \times (Z \cup \epsilon)$.

$(s, er, z, s', n, h) \in T$ means that there is a transition from state s to state s' with er on the input and z onto the stack. This transition pops z from the stack, then pushes h onto the stack. The transition is labeled with the notification n. We mark $er : z(n : h)$ on the transition.

In the Figure 5, we give an example of the compiled model. Only one notification is represented, indicating the *entry of an instruction in the second stage of the pipeline*. Two external resources are present. The second external resource allows the instruction `a`, with PC pc_d, to enter the pipeline.

The stack is used to choose the transition to fire in the particular case of function returns. The transition corresponding to the function call pushes onto the stack an identifier. When a function return instruction enters in the pipeline, the value is popped from the stack and determines the transition to fire. An example is given in Figure 6.

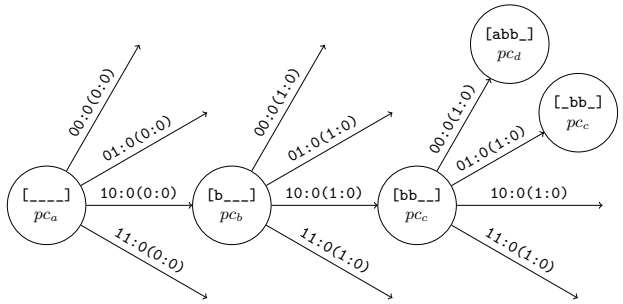

Figure 5: Automaton in compiled simulation: `10:0(1:0)` means that the first external resource is free and the second taken (`10:0(1:0)`, starting from least significant bit), that nothing is popped from the stack `10:0(1:0)`, that the notification happens (`1:0`) and that nothing is pushed onto the stack (`1:0`).

The management of branches is the same than in [2]: a specific external resource is allocated for this purpose. If the resource is taken, the branch is taken too, and conversely. It is necessary to use an external resource, because, in the general case, the branch condition can only be computed at runtime. The simulator is able to determine if a PC jump happens, then it can define dynamically the value of this resource. Using the same logic, a specific external resource can be used to represent the branch delay, according to the branching policy.

The example of Figure 7 shows the management of branches. The first external resource is allocated to manage branches. In particular, it is used for the branch `b` at PC pc_a. If this resource is not available, then the automaton goes to the target PC (pc_c), else it goes to the next PC (pc_b). The second resource is allocated to manage the branching latency. No instruction can enter the pipeline, while it is taken.

The construction of the automaton is quite difficult because the state of the system may be different when a function is called and depends on the caller. Consequently a simple algorithm based on the Breadth First Search is not sufficient. So when the program calls a function that have already been processed, the algorithm finds the previously created state. In this case, the algorithm jumps directly to the state modeling the function return, in order to continue the construction. This manipulation requires to record some information on the structure of the automaton, especially, for each state, which state models the previous function call, and which states model the next function returns. The algorithm is presented in the Figures 8, 9 and 10.

5. MACRO-INSTRUCTIONS

An important part of the execution time is devoted to the management of the automaton which is done at each clock cycle: one transition is fired to simulate one cycle. In this section, we propose a technique to speed up this task.

Thanks to the compiled simulation, it is possible that the automaton handles not only one instruction by transition, but a block of many instructions. We use the term of macro-instruction for this block. Then, firing one transition allows to simulate more than one cycle, speeding up the execution

label	pc	inst
	pc_a	call f
	pc_b	
		...
	pc_m	call f
	pc_n	
		...
f	pc_y	
	pc_z	return

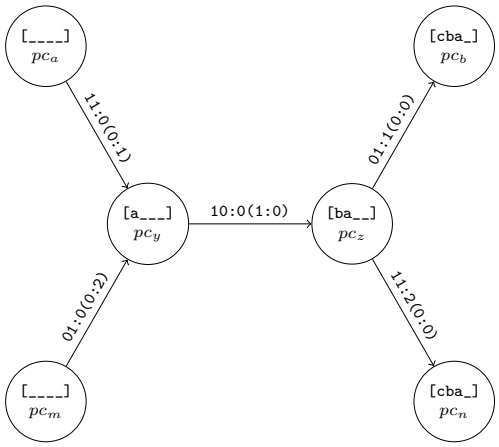

Figure 6: Management of the stack: `11:0(0:1)` means that 1 is pushed onto the stack, `01:1(0:0)` means that the transition is fired if 1 is popped from the stack. `10:0(1:0)` means that nothing is pushed onto the stack and `10:0(1:0)` means that nothing is popped. If the automaton comes from pc_a, at the return function, 1 is popped from the stack, then the automaton goes to pc_b.

of the simulation. A sequence of transitions can be reduced to one transition only, containing the concatenation of the information of the elementary transitions. We call a macro-transition a transition modeling more than one cycle. This reduction is only possible with linear sequence of transitions, as shown in Figure 11.

Indeed, if a state has several successors, conditions used to choose the correct following state should be done at runtime, as it depends on other hardware devices (a memory access for instance). Such a state is at the end of a linear sequence of instruction and terminates a macro-instruction.

The efficiency of this reduction depends on the rate of linear sequences of transitions in the automaton. These sequences appear only if the use of external resources is infrequent. Indeed, an external resource leads to two successors: if the external resource is taken or free, the evolution of the pipeline is blocked or not. In order to make the technique efficient, it is necessary to treat statically the behavior of some external resources particularly used. This treatment is allowed by the compiled simulation, as it is explained in [2].

One external resource of primary interest is the one related to the instruction fetch. As the instruction fetch requires a memory access, an external resource is used to model the fact that the memory is available or not. If the memory is

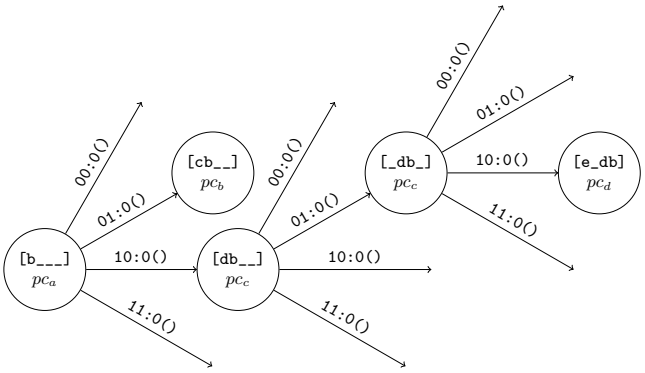

Figure 7: The second external resource specifies if a branch (like `b`) is taken or not. The first external resource is used to model branching latency (delaying in this case the entry of instruction `e` in the pipeline).

```
1: Initialisation
2: while list of states to process is not empty do
3:    s = pop(list of states to process)
4:    Find successors from state s
5: end while
```

Figure 8: Global program

not available, a pipeline stall is inserted into the pipeline. This external resource has a deep impact on the macro-instruction size because each instruction requires it.

However, this particular external resource related to instruction fetch modeling can be optimized if we consider the instruction cache. During the compilation process, we can test if the instruction currently processed is on the same instruction cache line than the previous instruction: 2 instructions are on the same cache line if their address differs only in the 5 least significant bits for a cache line of 32 bytes for instance. Obviously, if the instruction is on the same cache line, we can statically deduce that the cache access will be a *hit* and remove the external resource for this access. We can notice that this modification leads to an average of 70% reduction of automaton's states.

The integration of macro-instructions needs no deep modifications in the model. It is only necessary to manage the concatenation of information labeled on transitions and to add one: the number of cycles represented by the macro-transition. Since the macro-transition is the reduction of linear sequences of transitions, intermediate conditions bring no additional information to handle the automaton. Then, we only keep the first of them. Moreover, for implementation reasons and because such a condition is not really restrictive, we require that macro-transitions cannot own more than one *pop* and more than one *push*. To handle the automaton, it is not useful to execute this two actions at a precise moment. It allows us to bring backward their treatment at the beginning of the macro-transition. However, it is important to respect their chronology.

We can now formalize the model. Let MA be an automaton defined by $\{S, s_0, IC, ER, NC, P, N, T\}$, where:

- S is the set of states;

```
1: for all external resources do
2:    if instruction is a taken branch then
3:       if instruction is an indirect branch then
4:          Search the called state
5:          Add the return state in the called state
6:          for all call found in the called state do
7:             Computation of successor for call
8:          end for
9:       else
10:         Computation of successor for the target
11:      end if
12:   else
13:      Computation of successor for the next instruction
14:   end if
15: end for
```

Figure 9: Algorithm to find successors

- s_0 is the initial state (empty pipeline, initial PC) in S;
- IC is the first alphabet of actions (indirect condition);
- ER is the second alphabet of actions (external resources);
- NC is the first alphabet of labels (number of cycles);
- P is the second alphabet of labels (push);
- N is the third alphabet of labels (notifications);
- T is the macro-transition function in $S \times IC \times ER \times NC \times P \times N^* \times S$.

In the Figure 12, we give an example of the construction of a macro-transition. We suppose that the first transition pops ic_1 from the stack, and that the second transition pushes p_2 onto the stack. The middle state has only one successor. Then, conditions are gathered to construct a macro-transition. We replace all input transitions by the concatenation of itself and the transition to the successor. The macro-transition contains the concatenation of the two notifications, the indirect condition of the first transition, and the push value of the second transition. The numbers of cycles are added up. If $(s, ic, er, nc, p, n, s') \in T$, we mark $er : ic(n : p : nc)$ on the transition from s to s'.

The execution of the automaton is done as follows: The simulator computes external resources and indirect condition to handle the automaton. It finds the successor and fetches the information on the macro-transition. For each cycle modeled by the macro-instruction, the simulator decodes, executes the instructions and get notifications from the pipeline model.

The automaton is constructed as shown in the Figure 13. The condition to concatenate is: $(ic_1 = 0 \vee ic_2 = 0) \wedge (p_1 = 0 \vee (ic_2 = 0 \wedge p_2 = 0))$

6. TESTS AND PERFORMANCE

In this section, we present tests and performance of our model, in comparison with the interpreted simulation.

In these tests, we simulate a similar architecture to PowerPC 5516 from Freescale, with a *e200z1* core. The pipeline is resized from 4 to 5 stages to increase the size of the model. We ran benchmarks of Mälardalen [5]. Simulations are made

```
1: Creation of the successor
2: if instruction is a taken branch and is entered in the pipeline then
3:    if instruction is an indirect branch then
4:       Search of the call state corresponding with the return
5:    end if
6:    if instruction is a call then
7:       Record of the current index in the successors
8:       if successor does not exist then
9:          Initialisation of the structure
10:      else
11:         Add information in the structure
12:      end if
13:   end if
14: end if
15: if successor exists and does not point on the same state then
16:   Call state point on the first call state
17:   Fusion of information in the first call state
18:   Computation of successors for all possible returns
19: end if
20: Creation of the transition
21: Push state in the list of processed states
22: if successor is not included in the list of processed states or states to process then
23:   Push state in the list of states to process
24: end if
```

Figure 10: Algorithm to compute a successor

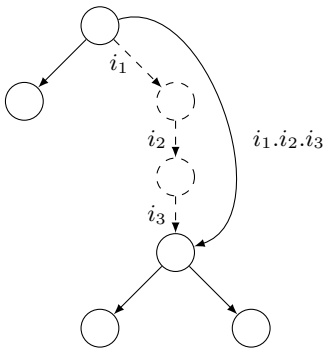

Figure 11: $i_1.i_2.i_3$ is called a macro-instruction. It is delimited by two branches.

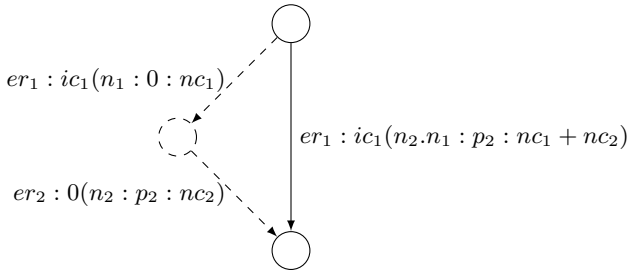

Figure 12: The construction of a macro-transition

with an Intel *Core i7@3,4GHz* computer. We execute 50 000 times each program.

The new model that we propose in this paper allows to reduce the automaton's size in comparison with the previous version of ComCAS model. We give in Table 1 the gain of this reduction. In [2], a calculus gives a theoretical value for the number of states, if the function call stack is not used. It is exactly the same result that we get, here. We can note that some programs have the same number of states with the two models. The reason is that the PC stack is not fully used: functions are called once during the execution. For the special case of programs using software floating point computation, the reduction is considerable, as we can see with the program basicMathVerySmall.

The use of macro-instructions leads to a reduction of the model. Macro-transitions are mainly limited by forks in the automaton. These forks are caused by branches and external resources. In the best case, *i.e.*, for a linear control flow with no data dependencies, a fork appears every time an instruction cache line boundary is crossed. In our example, a line of the instruction cache contains 8 instructions. Consequently, the reduction is bounded by 87,5%. In the Table 2, we observe an average reduction of 47,1%.

In the Figure 14, performance of our new ComCAS model is presented in comparison with the interpreted method. We observe a reduction of the execution time of 53% on average.

The model from [2] leads to a 45% decrease of the execution time. However, the simulation of programs using software floating point computation was impossible. With this new model, it becomes possible. The technique of macro-instructions reduce the time devoted to the handle of the automaton. It leads to improve the decrease of the execution time to 53% at average, and up to 57% at best, as we can see on Figure 14. The Figure 15 shows a wide comparison between execution time, especially with ISS. This comparison shows the reduction on the CAS specific part, and what improvement are still possible without speeding up the ISS. This benefit shows the interest for techniques that allows the compiled simulation in the validation of real-time embedded systems.

7. CONCLUSION

The contribution presented in this paper brings new techniques to implement high speed Cycle Accurate Simulator. We use a Pushdown Automaton in our model in order to extend the Compiled Cycle Accurate Simulator to programs using a large number of functions, like software floating point computation. Moreover, we have implemented a new technique: the use of macro-instructions. It consists in the gathering of instructions in a single block. We have compared performance of our model with the associated interpreted method. This leads to an average of 53% decrease of execution time in comparison with the interpreted simulator.

Future works will be about the use of the ComCAS model in a Just In Time simulator. This technique could bring a solution for the different problems of the compiled simulation, especially indirect branches. The Just In Time simulator would run like the interpreted simulator and would reduce the automaton during the execution stage, *on the fly*. When a loop would encountered, the simulator would switch to a reduced automaton, in order to improve performance

```
1: while concatenations done do
2:   for all state s do
3:     Account of successors of s
4:     if only one successor to s then
5:       for all input transition t of s do
6:         if condition of concatenation then
7:           Concatenation of t to the successor of s
8:         end if
9:       end for
10:    end if
11:  end for
12: end while
```

Figure 13: Algorithm for the construction of macro-transitions

Program	States with inlining	States with PA	Gain
adpcm	18 229	12 096	33,6%
basicMathVerySmall	322 615	4 302	98,6%
bs	587	587	0%
compress	5 901	5 059	14,3%
cover	1 123	1 123	0%
crc	3 616	2 038	43,6%
duff	669	669	0%
expint	1 395	1 395	0%
fdct	3 707	3 707	0%
fibcall	479	479	0%
fir	1 016	1 016	0%
janne_complex	645	645	0%
jfdctint	3 134	3 134	0%
lcdnum	568	568	0%
matmult	1 941	1 516	21,9%
ndes	8 378	6 114	27%
ns	837	837	0%
prime	1 871	1 138	39,2%

Table 1: Influence of the inlining on the size of the model

dynamically.

8. REFERENCES

[1] F. Bellard. Qemu, a fast and portable dynamic translator. *Translator*, 394:41–46, 2005.

[2] A. Bullich, M. Briday, J.-L. Béchennec, and Y. Trinquet. Comcas, a compiled cycle accurate simulation for hardware architecture. *The Fifth International Conference on Advances in System Simulation (SIMUL'13)*, October 2013.

[3] C. Cifuentes and V. Malhotra. Binary translation: Static, dynamic, retargetable? *Proceedings International Conference on Software Maintenance*, pages 340–349, 1996.

[4] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. *EDTC'95: Proceedings of the 1995 European Conference on Design and Test*, pages 503–507, March 1995.

[5] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In B. Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, July 2010. OCG.

[6] G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: an instruction set description language for retargetability. *DAC'97: Proceedings of the 34th annual conference on Design automation*, pages 299–302, 1997.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach-Second Edition*. Morgan Kaufmann Publishers, Inc., 2001.

[8] D. Jones and N. Topham. High speed cpu simulation using ltu dynamic binary translation. *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers*, pages 50–64, January 2009.

[9] R. Kassem, M. Briday, J.-L. Béchennec, G. Savaton, and Y. Trinquet. Simulator generation using an automaton based pipeline model for timing analysis. In *International Multiconference on Computer Science and Information Technology (IMCSIT'08)*, pages 657–664, Wisla, Poland, October 2008.

[10] R. Kassem, M. Briday, J.-L. Béchennec, G. Savaton, and Y. Trinquet. HARMLESS, a hardware architecture description language dedicated to real-time embedded system simulation. *Journal of Systems Architecture* -

Program	# states no macro-inst.	# states with macro-inst.	Gain
adpcm	12 096	5 729	52,6%
bs	587	317	46%
basicMathVerySmall	4 302	2 588	39,8%
compress	5 059	2 370	52,8%
cover	1 123	599	46,7%
crc	2 038	1 060	48%
duff	669	378	43,5%
expint	1 395	764	45,2%
fdct	3 707	1 984	46,5%
fibcall	479	240	49,9%
fir	1 016	574	43,5%
janne_complex	645	356	44,8%
jfdctint	3 134	1 727	44,9%
lcdnum	568	279	50,9%
matmult	1 516	863	43,1%
ndes	6 114	3 239	47%
ns	837	471	44,4%
prime	1 138	566	50,3%

Table 2: Influence of macro-instructions on the size of the model

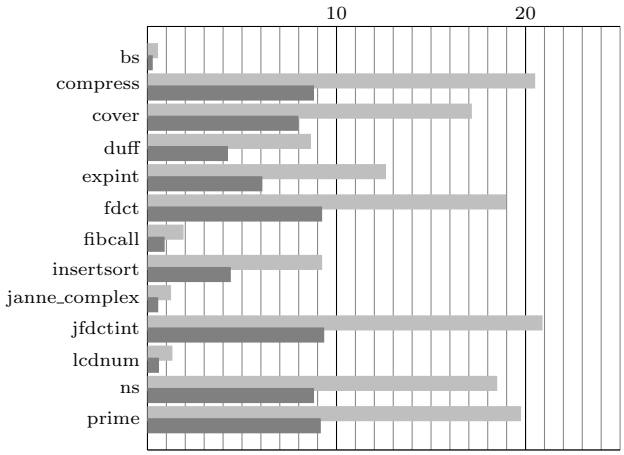

Figure 14: Comparison of execution time in seconds for 50 000 executions. Gray is for interpreted simulation, and black is for compiled simulation.

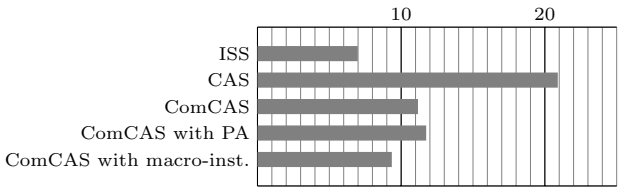

Figure 15: Comparison of execution time of different simulation's technique in seconds for jfdcint. If we only consider the CAS specific part, our model leads to a 83% reduction.

doi: http://dx.doi.org/10.1016/j.sysarc.2012.05.001 [retrieved: august, 2013], pages 318–337, September 2011.

[11] R. Kassem, M. Briday, J.-L. Béchennec, Y. Trinquet, and G. Savaton. Instruction set simulator generation using HARMLESS, a new hardware architecture description language. *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 24:1–24:9, 2009.

[12] P. Mishra and N. Dutt, editors. *Processor description languages*. Morgan Kaufmann Publishers, 2008.

[13] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. Lisa - machine description language for cycle-accurate models of programmable dsp architectures. *DAC'99: Proceedings of the 36th ACM/IEEE conference on design automation*, pages 933–938, 1999.

[14] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, pages 47–56, 2004.

[15] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–95, June 2003.