# ComCAS: A Compiled Cycle Accurate Simulation for Hardware Architecture

Adrien Bullich, Mikaël Briday, Jean-Luc Béchennec and Yvon Trinquet
IRCCyN - UMR CNRS 6597
Nantes, France
Email: {first name.last name}@irccyn.ec-nantes.fr

***Abstract*—This article is in the context of real-time embedded systems domain. These critical systems require an important effort in validation and verification that can be done at many abstraction levels, from high-level application model to the actual binary code using an accurate model of the processor. As the development of a handwritten simulator of a processor at a cycle accurate level is a difficult and tedious work, we use HARMLESS, a hardware description language that can generate both a functional and a cycle accurate simulators. The latter gives a temporal information of the simulation execution, but at the cost of a heavy computation overhead. This paper applies the compiled simulation principles to a cycle accurate simulator. It shows that this simulation mechanism can reduce computation time up to 45%, preserving timing information.**

***Keywords*—Cycle Accurate Simulation; interpreted simulation; compiled simulation; HADL**

## I. INTRODUCTION

Verification of a real-time application is a huge and difficult problem. It must be led throughout the development cycle on functional and extra-functional aspects (temporal aspects, safety, etc.). Our work takes place in the last stages of the development process, when the actual binary code of the application is available, just before the final test on the real target.

A simulation scheme should be chosen according to the studied field, pursued objectives, and the abstraction level required. The lower is the abstraction level, the higher is the simulator complexity. For hardware simulation, the implementation is complex, time consuming, and errors are difficult to avoid. To alleviate this complexity, a Hardware Architecture Description Language (HADL) may be used. With such a language, the complexity remains partly hidden. For the work presented herein, HARMLESS [1] has been used to build the simulators. The HARMLESS compiler can generate both functional simulators, *i.e.*, Instruction Set Simulator (ISS), and temporal simulators, *i.e.*, Cycle Accurate Simulator (CAS) from a common description. We consider here especially CAS, as timings of the application should be taken into account for real-time systems.

A CAS simulator requires much more computation time than an ISS. In [1], the CAS is about 7 times slower than the functional one on a simple PowerPC processor with a 5-stages pipeline. So, CAS related computation has room for improvement. Here, we propose to explore a technique to improve the speed of CAS, which is a compiled simulation for CAS.

The paper is organized as follows: Section II presents the related works; Sections III and IV explain the current model of interpreted simulation and the new approach using the compiled simulation; Section V evaluates the model size and Section VI presents some results on a set of benchmarks; Section VII concludes this paper.

## II. RELATED WORKS

Many HADLs have been proposed in the literature. Some of these HADLs only focus on the functional aspects of the instruction set they describe. So, the associated toolset is only able to generate an ISS. nML [2] and ISDL [3] are examples of this kind of HADLs. Other HADLs add a micro-architecture description from which a temporal behavior is constructed. For instance, LISA [4], MADL [5] and HARMLESS [6], [7] have the ability to generate an ISS and a CAS.

An interpreted simulator simulates the execution of a binary executable by doing the same steps as the hardware it simulates. So, for each binary instruction an ISS does the following steps: instruction fetch, instruction decode, and instruction execution. In addition, a CAS computes instructions dependencies, controls concurrent accesses to the buses, register files, and generally any computing resource of the architecture.

A compiled simulator is customized to execute a particular binary executable. Knowing the binary executable at the compilation stage, it allows to remove from the execution stage all the tasks that depend on the executed instruction only. As a result, a compiled simulator exhibits better performance than an interpreted one, but it has a longer compilation time. Since compilation is done less times than execution, classically one compilation for several executions, a compiled simulator offers a global gain of time. However, a compiled simulator is less flexible because it is attached to a particular program: if the user wishes to simulate another program, he needs to compile again.

For an ISS, compiled simulation consists in Binary Translation (BT) ([8] or [9]). First, the binary executable one wants to simulate is translated to a native binary of the host simulation platform. Then, the native binary is executed on the host simulation platform.

For a CAS, few methods exist for compiled simulator generation. The technique of BT cannot easily be adapted to CAS, but solutions exist [10], coupling interpreted parts and translated parts. We also find statistic approaches, called Cycle Approximate Simulator, based on the sampling of instructions [11]. However, it is not exactly equivalent to a CAS, because of errors margin.

To the best of our knowledge, the technique of compiled simulation has not yet been employed to speed CAS up, because of the restrictions it implies: it is difficult to determine statically the evolution of the micro-architecture. However, this is the main contribution of the paper.

## III. INTERPRETED SIMULATION MODEL

The contribution of the compiled simulation must be assessed in comparison with the associated interpreted approach. In this section, we present the interpreted model of the Cycle Accurate HARMLESS-based simulator [1], that is the base of our ComCAS model.

Copyright (c) IARIA, 2013. 

In the interpreted model, instructions of the application code are decoded and executed during the simulation. The model of a cycle accurate processor includes the instruction set and the memory model for functional execution and all the micro-architecture related parts that alter timings, as presented in the development chain in Figure 1.

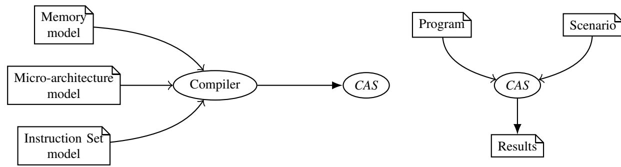


Fig. 1. The development of a CAS requires the modeling of the instruction set, the memory and the micro-architecture

One important micro-architecture unit is the processor pipeline: it has an influence on timings of the processor and this is the most expensive in computation time. Ideally, each instruction in each pipeline stage progresses to the next stage at each processor cycle. Actually, an instruction can be blocked in a pipeline stage because of *hazards*. Hazards are classified into three categories [12]: *structural hazards* are the result of 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*, which occur when a branch is taken in the program (one or more instructions that just follow the branch according to the branch delay that should be flushed). When a hazard is encountered, it is solved by stopping a part or all parts of the pipeline. This is called a *pipeline stall*.

Sequential pipelines are considered in this paper (*i.e.*, there are neither pipelines working in parallel, nor forking pipelines). The pipeline behavior is modeled in HARMLESS using an automaton, where a state represents the pipeline state at a particular time (see Figure 2).

In [1], the authors use the model of finite automata, because the system can be considered as a discrete transition system, a transition being taken at each cycle, as in Figure 2. The contribution of this paper is based on this definition. A state represents the system in a particular cycle. A state is defined by:

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

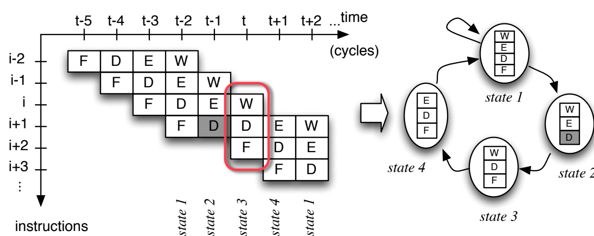


Fig. 2. A state of the automaton represents the state of the pipeline at a given time. Here, the pipeline has 4 stages. F: instruction is fetched, D: instruction is decoded and registers are read, E: instruction is executed, and W: the result is written into a register.

*Internal resources* are elements of the micro-architecture that are used only by the pipeline. Their availability allows or not the progression of an instruction in the pipeline. Stages of the pipeline themselves are considered as internal resources.

Instructions that use the same resources in the same pipeline stage are grouped together to form *instruction classes*. This is the case for instance for arithmetic instructions that read two registers, make a calculation, and write the result into a third register. Since internal resources depend only on the instruction class and on the pipeline stage, they are not needed at run time.

As a result, a transition represents a discrete event that brings the system from a state to another. It is determined by the state of *external resources* and the next instruction class that enters the first stage of the pipeline.

*External resources* are elements that are not used only by the pipeline, *i.e.*, their state is defined in other micro-architecture parts such as memory caches. The availability of these external resources has an influence on the evolution of instructions in the pipeline, too. Moreover, as they are external of the pipeline model, their availability is determined during the execution.

The content of states is abstracted, and information required for the simulation is gathered on transitions. For this reason, transitions are labeled with *notifications* (signaling if a particular event happens or not).

We can now formalize the model. Let $AI$ 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 $S \times ER \times IC \times N \times S$.

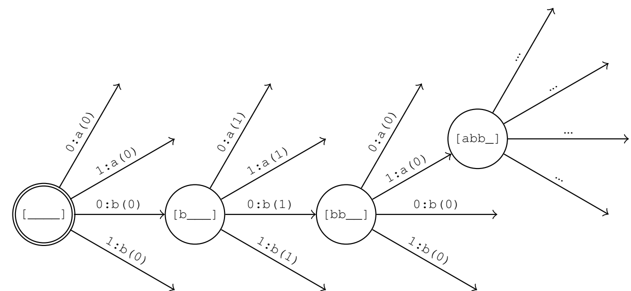


Fig. 3. Automaton in interpreted simulation: `0:b(1)` means that the external resource is free (`0`), that the instruction `b` may enter the pipeline and that the notification happens (`1`)

In the example of Figure 3, the notation `[b_a_]` represents the state of the 4-stages pipeline: it means that instruction of class `b` is in the first stage and instruction of class `a` is in the third stage. There are no other instruction classes for readability. We have only one notification that represents the *entry of an instruction in the second stage of the pipeline*. There is one external resource. The instruction class `b` needs to take the external resource to enter the pipeline.

During the simulation, both the state of external resources and the instruction class of the next instruction that will enter the pipeline are required to determine the next state of the automaton. When a transition is taken, notifications related to the transition are given to the simulation engine to interact with other micro-architectural parts.

## IV. ComCAS model

In this section, we adapt the interpreted model to be a compiled one: the ComCAS model.

The compiled simulation differs from the interpreted simulation in the repartition of tasks between compilation and execution. We recall that, in our case, the task is the analysis of the program. An interpreted simulator analyzes the program during the execution. A compiled simulator analyzes the program during the compilation.

Because of this change, the compiled simulation has a faster execution than the interpreted simulation. However, the compiled simulation has a longer compilation time than the interpreted simulation. This is not necessarily a problem: usually, the compilation is done only once, while the execution is performed several times.

A compiled simulation is run for a special architecture and for a special program. Consequently, the simulator is less flexible, attached to a particular program. If the need is to simulate several programs, the interpreted simulation will be more efficient. But, if the need is to simulate only one program with different scenarios, the compiled simulation will be more efficient.

In Figure 1 and Figure 4, we can see the difference between the development chain of the interpreted simulation and the compiled simulation, respectively. We notice especially that for the interpreted simulation the program is at the end of the development chain and at the beginning for the compiled simulation.

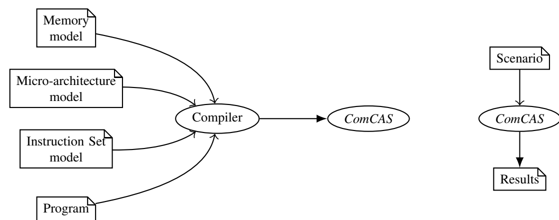


Fig. 4. The development of a compiled CAS requires to move the program analysis in the compilation

To transform the interpreted model into a compiled model, we need to add some information about the program. We first need its memory mapping, *i.e.*, the location of each instruction in memory, the corresponding Program Counter (PC) and the stack of called function (a stack of PC, in order to return to previous functions). Then, the determination of our system is given by:

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

With this model, instructions become labels on the automaton and no more actions are needed. Indeed, we only determine the evolution of the system with external resources and instructions become an information we get out of this run. However, it cannot be reduced to a simple notification (a boolean information), so we add the PC on the transition label.

We can formalize our ComCAS model as it follows.

Let $AC$ be an automaton defined by $\{S, s_0, ER, I, N, T\}$, where:

- $S$ is the set of states;
- $s_0$ is the initial state (empty pipeline, initial PC, empty stack) in $S$;
- $ER$ is the alphabet of actions (external resources);
- $I$ is an alphabet of labels (instructions);
- $N$ is an other alphabet of labels (notifications);
- $T$ is the transition function in $S \times ER \times I \times N \times S$.

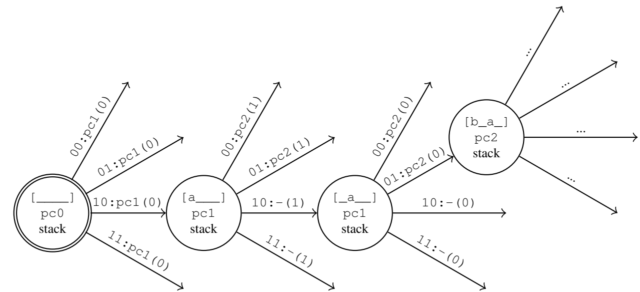


Fig. 5. Automaton in compiled simulation: `10:pc1(0)` means that the first external resource is free and the second taken (`10`), that the instruction with PC `pc1` enters the pipeline and that the notification does not happen (`0`)

In the example from Figure 5, we have only one notification that represents the entry of an instruction in the second stage of the pipeline. There are two external resources. The instruction `b`, with PC `pc2`, needs to take the second external resource to enter the pipeline.

The management of branches uses a specific external resource. If this resource is taken, the branch is taken and conversely. The use of an external resource is mandatory because in the general case, the branch target can only be computed at runtime. During the simulation, we can detect if a branch is taken and define dynamically the value of this resource. In order to represent the latency of the branching, according to the branching policy, another specific external resource could be employed to model *control hazards*. If the micro-architecture uses a branch predictor, the simulator would emulate this branch predictor and define dynamically the value of the corresponding external resource. While the resource is defined to be taken, the instruction that follows could not enter the pipeline.

An example is given in Figure 6. The first external resource represents the branch management (used in this case for the branch `b` to `pc3`). If it is taken, then the model goes to the target PC (`pc3`), else it goes to the next PC (`pc1`). The second resource represents the branching latency. As long as it is taken, no instruction can enter the pipeline.

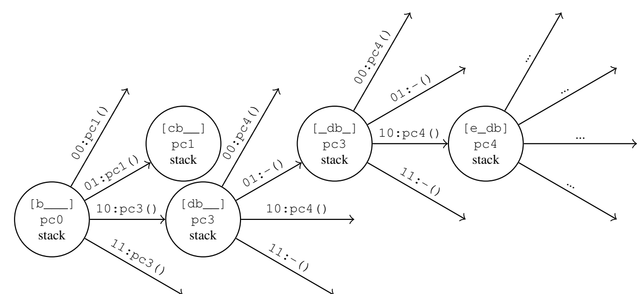


Fig. 6. The second external resource specifies if a branch (like `b`) is taken or not. The first external resource is used to model branching latency (delaying in this case the entry of instruction `e` in the pipeline).

The compiled simulation needs to compute statically the control flow of the program, in order to model it during the compilation. In the case of indirect branches or code self-modifying, this computation is impossible unless we execute the different scenarios of the program. This is the main restriction of our model. Without the possibility to determine statically the target of indirect branches, the solution is to plan the different possibilities. Unfortunately, it leads to a considerable increase of the automaton's size.

Indirect branches are unavoidable if we want to model functions calls, because of `RETURN` instructions. This specific problem is solved with a PC stack that is added in states. The stack in states allows to memorize original PC when a `CALL` instruction is executed. When a `RETURN` instruction is executed, this stack allows to determine the target PC of the branch, during the compilation.

The process is the following: when a `CALL` instruction enters the pipeline, we push the next PC onto the stack, and we branch to the target PC. When a `RETURN` instruction enters the pipeline, we pop a PC from the stack, and we branch on it. An example is given in Figure 7.

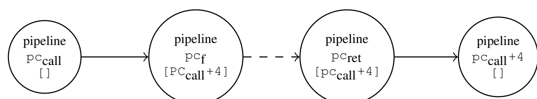


Fig. 7. A `CALL` function pushes the original PC onto the stack, and a `RETURN` function pops the PC from the stack

The main advantage of the compiled simulation is to move a computation part from the runtime to the compile time. This is the case for *data hazards* that are handled in the interpreted simulator using an external resource. This is a costly task that could be solved at compile time in ComCAS: the instructions in the pipeline are known, so we can determine all the registers that are read and written statically.

## V. Valuation of the number of states

In order to give an idea of the complexity of ComCAS model, we propose to evaluate the number of states.

A state is composed of the pipeline state, the PC corresponding to the last instruction read and the stack of called functions. In a first step, we will consider there is no stack in states, and we will add this feature afterwards. The global method consists in counting pipeline states for a given PC. We find three different situations in the control flow: linear configuration, beginning of the program and branching configuration.

In a linear configuration, for a given PC, one past exists. Consequently, the state of the pipeline is only determined by pipeline stalls. The problem is reduced to a combinatory one: if $s$ is the number of stages, we count $C_s^k$ ($k$ among $s$) possible pipeline states with $k$ instructions inside ($k \in [0; s]$). Thus, the total number of pipeline states is $\sum_{k=0}^{s} C_s^k$.

If the PC points at the beginning of the program, $pc_n$ with $n < s$, it is impossible to put more than $n$ instructions in the pipeline. So, in this case, the previous value is truncated to $\sum_{k=0}^{n} C_s^k$. To simplify computation, from now on, we use $f_s : n \to \sum_{k=0}^{n} C_s^k$. And we know that $f_s(s) = 2^s$.

At this step of our computation, we can valuate the number of states in a perfect linear program (with no branch). Let $i$ be the number of instructions. The first $s$ instructions are in the second case (beginning of the program), and others $i - s$ instructions are in the first case (endless linear configuration). It gives: $\sum_{k=0}^{s-1} f_s(k) + (i - s).2^s$.

This value is a maximum, and it is reached if every pipeline states is explored. It is the case when an external resource manages the entry of instructions in the first stage (bus access or cache miss), allowing all stalls arrangements.

The number of pipeline states is larger if we include branches in the control flow. Let us consider the case in Figure 8, with $k < s$. In this situation, if we put $j$ instructions in the pipeline with $j \leq k$, the branch is not visible in the pipeline. Thus, we remain in the same previous situation: $C_s^j$ pipeline states. But, if we put $j$ instructions in the pipeline with $j > k$, then for each pipeline stalls arrangement two pipeline states exist, with two different pasts. So, we count $2.C_s^j$ pipeline states. The total number of pipeline states is $\sum_{j=0}^{k} C_s^j + \sum_{j=k+1}^{s} 2.C_s^j$ It is equivalent with $2^{s+1} - f_s(k)$.

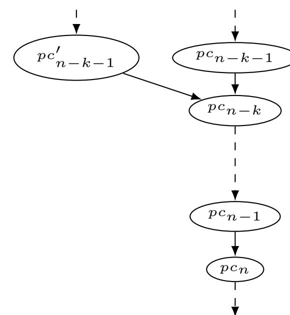


Fig. 8. A branch configuration in the control flow. If $k < s$ then in PC $pc_n$ the pipeline can remember two pasts.

For one branch situation, we have $k$ varying in $[0; s - 1]$. Let $b$ be the number of branch targets. So, the total number of states becomes:

$$\sum_{k=0}^{s-1} f_s(k) + (i - s - s.b).2^s + b.(s.2^{s+1} - \sum_{k=0}^{s-1} f_s(k)) \quad (1)$$

It is equivalent to:

$$(1 - b). \sum_{k=0}^{s-1} f_s(k) + (i - (1 - b).s).2^s \quad (2)$$

The expression is valid if branch configuration is the same as the one we give in Figure 8. It means that two conditions arrise:

- branch targets are separated by more than $s$ instructions;
- no branch is less than $s$ instructions after a branch target.

In fact, we can confirm that the first condition does not degrade our valuation. The second condition is more important and precludes too small loops.

The analysis of our valuation reveals that the number of states is linear with the number of instructions, and exponential with the number of stages. We can compare the expression with the number of states in interpreted simulation: $(ic + 1)^s$,

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-308-7 

which is more exponential considering $s$, $ic$ being the number of instruction classes.

In our valuation, we have not yet considered the use of PC stack in states. We can see in Figure 9 the effect on the control flow of the add of PC stacks.

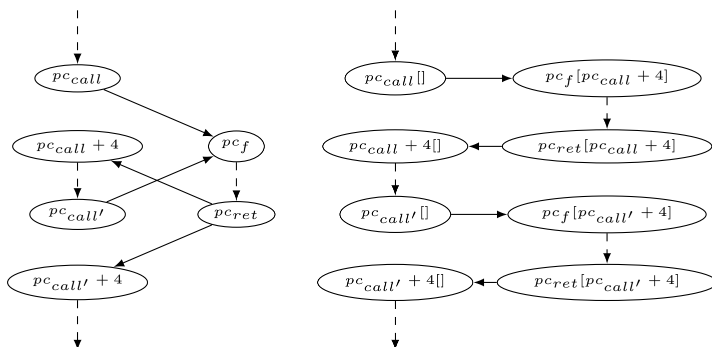


Fig. 9. Stack consideration consists in an inline in the control flow. These two automata are equivalent.

It can be regarded as inlining: functions' code is duplicated. If we consider this new control flow, our reasoning is the same. Let us call $i'$ the number of couple of instruction and PC stack, and $b'$ the new number of branch targets. Thus, the number of states simply becomes:

$$(1 - b').\sum_{k=0}^{s-1} f_s(k) + (i' - (1 - b').s).2^s \tag{3}$$

The influence of this inlining is very dependent on the code (see Table I). For example, we observe that programs using software floating point numbers increase significantly the size of the automaton, and make difficult the construction of the model.

## VI. Tests and Performance

In this section, we present experimental results about performance of ComCAS model in comparison with the interpreted simulation.

The architecture simulated in these tests is similar to a PowerPC 5516 from Freescale, with a *e200z1* core. The pipeline has been resized from 4 to 5 to increase the size of the model. We ran the benchmarks of Mälardalen [13]. Simulations are made with an Intel *Core i7@3,4Ghz* computer. We execute 50 000 times each program.

We give in Table I an illustration of the influence of the inlining and the number of states, obtained by ComCAS tool. This allows to confirm that if a function is called once during the execution, PC stack has no influence on the size of the model. We note that the number of states is smaller than the valuation we can compute, because the model does not explore every pipeline states. With particular external resources (making every pipeline states possible) we get the same result than our valuation. To allow a comparison, with the same configuration the interpreted model gets 1 024 states. Smaller is the code, smaller is our model.

Figure 10 represents the performance of ComCAS model in comparison with the interpreted method for the execution time. In the compiled approach, the generation of the simulator is more complex, as it requires to generate the ISS, analyze the program (using the ISS) and build the simulator. This time consuming compilation step is largely counterbalanced by a faster execution time, which is done several times.

TABLE I. Influence of the inlining: **i** is the number of instructions, **b** the number of branch targets, **i'** the number of instructions with the inlining and **b'** the number of branch targets with the inlining

| Program | i | b | i' | b' | States |
|---|---|---|---|---|---|
| adpcm | 2 243 | 79 | 3 308 | 79 | 75 588 |
| bs | 84 | 4 | 84 | 4 | 2 061 |
| compress | 867 | 40 | 1 027 | 43 | 24 586 |
| cover | 145 | 7 | 145 | 7 | 3 434 |
| crc | 322 | 11 | 584 | 19 | 13 022 |
| duff | 88 | 3 | 88 | 3 | 2 101 |
| expint | 185 | 8 | 185 | 8 | 4 544 |
| fdct | 692 | 3 | 692 | 3 | 14 638 |
| fibcall | 58 | 3 | 58 | 3 | 1 447 |
| fir | 144 | 5 | 144 | 5 | 3 398 |
| insertsort | 131 | 3 | 131 | 3 | 2 961 |
| janne_complex | 76 | 6 | 76 | 6 | 1 974 |
| jfdctint | 551 | 4 | 551 | 4 | 11 605 |
| lcdnum | 74 | 4 | 74 | 4 | 1 768 |
| matmult | 203 | 7 | 274 | 8 | 6 844 |
| ndes | 1 009 | 31 | 1 377 | 47 | 32 976 |
| ns | 116 | 8 | 116 | 8 | 2 907 |
| nsichneu | 12 511 | 626 | 12 511 | 626 | 275 322 |
| prime | 147 | 8 | 268 | 9 | 6 706 |

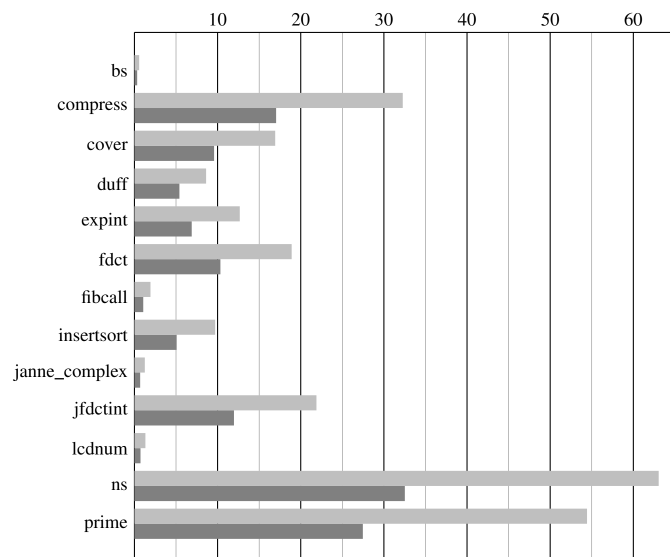


Fig. 10. Comparison of execution time in seconds for 50 000 executions. Gray is for interpreted simulation, and black is for compiled simulation.

The main impact of our model comes from the ability to manage analysis tasks when compiling. In particular, the treatment of the data dependency control in the compilation phase has been implemented in ComCAS. It reduces the execution time by 45,1% on average as we can see on Figure 10, and up to 49,5% with *prime* benchmark. This significant benefit shows the interest for the compiled simulation for the validation of real-time embedded systems.

## VII. Conclusion

In this paper, we have discussed the different techniques to implement high speed Cycle Accurate Simulator. We have

developed a model to adapt the compiled simulation approach to Cycle Accurate Simulator and implement it in the ComCAS tool. We have studied the maximum theoretical size of our model and compared performance of our model with the associated interpreted method. These results show that the computation time is reduced by 45% in comparison with the interpreted simulator.

Compiled simulation is efficient because it allows to remove some analysis tasks from the execution step. Even if this technique does not currently handle indirect branches, function calls are taken into consideration to simulate a major part of embedded systems programs.

Future work aims at improving the efficiency of the ComCAS model by using *macro-instructions*. A macro-instruction gathers the behavior and the timing of a set of successive instructions provided there is no external resource used by these instructions. However, an external resource attached to the *fetch* stage is needed and precludes the construction of macro-instructions. The solution could be to take the cache behavior into account to remove this external resource. With this improvement, the size of the automaton would be reduced and the performance of the simulator would be increased.

Another path of improvement would be to use the ComCAS model in a Just In Time simulator. In this case, the interpreted simulator would reduce the automaton *on the fly* when a loop is encountered and would switch its execution to the reduced automaton to improve performance dynamically. This could bring a solution for the problem of indirect branches.

## REFERENCES


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

[2] 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, March 1995, pp. 503–507.

[3] 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, 1997, pp. 299–302.

[4] 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, 1999, pp. 933–938.

[5] 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), 2004, pp. 47–56.

[6] 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), Wisla, Poland, October 2008, pp. 657–664.

[7] 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, 2009, pp. 24:1–24:9.

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

[9] F. Bellard, "Qemu, a fast and portable dynamic translator," Translator, vol. 394, 2005, pp. 41–46. [Online]. Available: http://www.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard_html/[retrieved:august,2013]

[10] 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, January 2009, pp. 50–64.

[11] 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, June 2003, pp. 84–95.

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

[13] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in WCET2010, B. Lisper, Ed. Brussels, Belgium: OCG, July 2010, pp. 137–147.


Copyright (c) IARIA, 2013. ISBN: 978-1-61208-308-7 