Towards a Parameterizable Cycle-Accurate ISS in ArchC

Charly BECHARA and Nicolas VENTROUX

CEA, LIST,
Embedded Computing Laboratory,
Gif-sur-Yvette, F-91191, FRANCE;
Email: charly.bechara@cea.fr

Daniel ETIEMBLE

Université Paris Sud,
Laboratoire de Recherche en Informatique,
Orsay, F-91405, FRANCE;
Email: daniel.etiemble@lri.fr

***Abstract*—With the increase in the design complexity of MPSoC architectures, flexible and accurate processor simulators became a necessity for exploring the vast design space solutions. In this paper, we present a flexible cycle-accurate ISS model based on ArchC 2.0 language. The model can have a variable pipeline depth and can be integrated easily in any SoC design based on SystemC. Its performance and capabilities are demonstrated by running MiBench embedded benchmark suite, while extracting pipeline statistics for each application.**

keywords: ISS, cycle-accurate, System-on-Chip, ADL, Design Space Exploration

I. Introduction

The emergence of new embedded applications for telecom, automotive, digital television and multimedia applications, has fueled demand for architectures with higher performances, more chip area and more power efficiency. These applications are usually computation-intensive, which prevents them from being executed by general-purpose processors. Thus, designers are showing interest in a System-on-Chip (SoC) paradigm composed of multiple processors and a network that is highly efficient in terms of latency and bandwidth. The resulting new trend in architectural design is the MultiProcessor SoC (MPSoC) [1]. MPSoCs' architectures can have homogeneous or heterogeneous processors, depending on the application requirements. Choosing the best processor among hundreds of available architectures, or even designing a new processor, requires the evaluation of many different features (pipeline structure, ISA description, register files, processor size...), and the architect needs to explore different solutions in order to find the best trade-off. The processor Instruction Set Simulator (ISS), which its role is very important, must have the following features: it should be parameterizable, fast and accurate, and be able to be integrated easily in the MPSoC simulation environment.

The ISS emulates the behavior of a processor by executing the instructions of the target processor while running on a host computer. Depending on the abstraction level, it can be modeled at the functional or cycle-accurate level. The functional ISS model abstracts the internal hardware architecture of the processor (pipeline structure, register files...) and simulates only the ISA. Therefore, it can be available in the early phase of the MPSoC design for the application software development, where the simulation speed and the model development time are an important factor for a fast design space exploration. Despite all these advantages, many details are hidden by the functional ISS model, such as the pipeline stalls, branch/data hazards and other parameters, which tend to be non-negligible while sizing the architecture. Those parameters evaluate the accurate performance of the processor and the surrounding hardware blocks such as caches, busses, and TLBs. The cycle-accurate ISS model simulates the processor at an abstraction level between the RTL and the functional model. It presents most of the architectural details that are necessary for processor dimensioning, in order to evaluate in advance its performance capabilities in the MPSoC design. All these advantages come at the expense of its slower simulation speed and longer development time.

The pipeline depth is one important parameter for processor sizing. A deeper pipeline using more pipeline stages allows a higher clock frequency. On the other hand, a deeper pipeline leads to increased load-use latencies, increased branch latencies and mispredicted branch penalties. In any case, multicycle instructions, such as integer multiplication and division and all the floating-point instructions are mandatory. The evaluation of processor performance when varying the number of execute stages in the processor pipeline cannot be avoided.

This paper investigates the ability of the cycle-accurate ISS model to be used as part of design space explorations. For this reason, we developed a variable pipeline depth model with pipeline statistics extraction.

The paper is organized as follows: Section II discusses related works on different types of ADLs and motivates the reason to chose ArchC language. Then, section III gives an overview of the R3000 cycle-accurate ISS in ArchC. Section IV describes the modifications done for the R3000 architecture, ISA description file and ArchC tool in order to generate a variable pipeline depth ISS model, while section V highlights the statistics and debugging logs that the model generates. The R3000 architecture is taken as an example and our approach can be easily deployed for other ISAs such as SPARC and ARM. Section VI illustrates the performance results obtained by running MiBench embedded benchmark suite [2], and compare them to those of the functional model.

Finally, section VII concludes the paper by discussing the present work along with future works.

II. Related Work

The main part of an MPSoC simulator is the architecture description language (ADL), which generates an ISS in a specific level of abstraction. ADLs' modelisation levels are classified into three categories: structural, behavioral, and mixed.

Structural or cycle-accurate ADLs describe the processor at a low abstraction level (RTL) with a detailed description of the hardware blocks and their interconnection. These tools, such as MIMOLA [3], are mainly targeted for synthesis and not for design space exploration due to their slow simulation speed and lack of flexibility.

On the contrary, behavioral or functional ADLs abstract the microarchitectural details of the processor, and provide a model at the instruction set level. Its low accuracy is compensated by its fast simulation speed. Many languages exist such as nML [4] and ISDL [5].

Therefore, mixed ADLs provide a compromise solution and combine the advantages of both the structural (accuracy) and behavioral (simulation speed) ADLs. It is the best abstraction layer for design space exploration. EXPRESSION [6], MADL[7], LISA [8], and ArchC[9] are an example of mixed ADLs. The last two will be discussed in this literature review.

LISA, which stands for Language for Instruction Set Architecture, is developed by the university of RWTH Aachen and is currently used in commercial tools for ARM and CoWare (LISATek). Processor models can be described in two main parts: resource and operation declarations (ISA). Depending on the abstraction level, the operations can be defined either as a complete instruction, or as a part of an instruction. For example, if the processor resources are modelled at the structural level (pipeline stages), then the instructions' behavior in each of the pipeline stages should be declared. Hardware synthesis is possible for structural processor models.

A recent type of processor description language called ArchC [10] is gaining special attention from the research communities [11], [12], [13]. ArchC 2.0 is an open-source Architecture Description Language (ADL), developed by the university of Campinas in Brazil. It generates from processor and ISA description files, a functional or cycle-accurate ISS in SystemC. The ISS is ready to be integrated with no effort in a complete SoC design based on SystemC [14]. In addition, the ISS can be easily deployed in a multiprocessor environment thanks to the interruption mechanism based on TLM, which allows the preemption and migration of tasks between the cores. The main distinction of ArchC is its ability to generate a cycle-accurate ISS with little development time. Only the behavior description of the ISA requires accurate description. As for the microarchitectural details, they are generated automatically according to the architecture resource description file. There exists also a graphical framework, called PDesigner [15], based on Eclipse and ArchC processor models, which allows the development and simulation of MPSoCs in SystemC in a friendly manner. Since ArchC is an open-source language, we can modify the simulator generator to produce a processor with customized microarchitectural enhancements, which makes it a great tool for computer architecture research [16]. However, the processor model cannot be synthesized because it is not supported by ArchC.

In this work, we provide a parameterizable cycle-accurate ISS model based on MIPS-I ISA as an example. We modified ArchC 2.0 to generate the model, which is ready to be integrated in a multiprocessor SystemC environment. In its first version, we support the variation of the number of EX stages in the pipeline, without model regeneration and recompilation. Processor performance evaluation is done through the extraction of pipeline statistics such as the number of stalls, their penalties and their types (branch/data hazards, memory access). This will provide the architects new parameters to dimension the processor according to the target design, which was not possible before with functional model processors.

III. Overview of the R3000 cycle-accurate model

The MIPS-I R3000 architecture is a classic 5-stage RISC processor (IF-ID-EX-MEM-WB) with 32 registers and an integer pipeline. The implemented MIPS-I ISA is similar to the optimized version described in [17]. The control instructions (jump and branch) are executed in the ID stage instead of the MEM stage, and follow the "predicted-not-taken" branch mechanism. Register forwarding is also deployed to allow instructions in the ID or EX stages to get the correct operand values from instructions that are further in the pipeline and did not commit yet. Both techniques reduce the number of pipeline stalls at the expense of adding more logics in the processor datapath.

ArchC 2.0 provides many advantages that lacked in its predecessor ArchC 1.6. First of all, it allows the simulator to be integrated and instantiated multiple times in a full SystemC platform, hence enabling multiprocessor system simulation. Second, the simulator is wrapped by a TLM interface to permit processor interruption and TLM communications with external modules. Finally, the functional 'acsim' and cycle-accurate 'actsim' simulator generators are implemented separately, which eases the development task.

Both functional and cycle-accurate processor models exist in ArchC 2.0 [9], and they are generated by a separate Simulator Generator tool. For instance, 'actsim' tool generates the cycle-accurate simulator. It parses the architecture resource description (AC_ARCH) and ISA description (AC_ISA) files, and generates the cycle-accurate simulator and the decoder accordingly, as illustrated in Figure 1.

Note that the resource and ISA description files must be described differently for the functional simulator. It is clearly seen that the cycle-accurate simulator is almost similar to the actual processor architecture. The pipeline stages, pipeline registers, register files, PC, and clock are all included in the simulator.

In our work, we utilize the latest available versions of 'actsim' timed simulator generator tool included in the ArchC

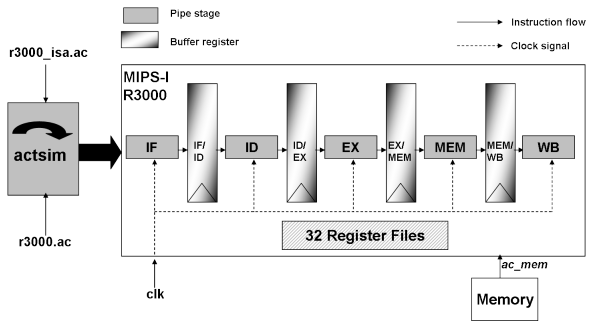

Figure 1. R3000 cycle-accurate model generation by actsim tool

2.0 package, as well as the MIPS-I R3000 cycle-accurate model (r3000-v0.7.2-archc2.0beta3). Both tools are still in their beta versions as they contain some bugs. In other words, the advantages of ArchC 2.0 have not been integrated in the cycle-accurate simulator. Using 'actsim' and R3000 model will allow the exploration of the cycle-accurate ISS performances, and the implementation of our architectural modifications, realized through the variable pipeline depth cycle-accurate processor model.

IV. Tools Modifications

In this part, we show the modifications done for the cycle-accurate simulator generator tool 'actsim' and the MIPS-I ISA implementation. Those modifications are necessary for the generation of a variable pipeline depth model, which can be integrated in a SoC design based on SystemC.

A. Modifications for ArchC 2.0

The initial 'actsim' generates, for each pipeline stage, a corresponding SystemC module, which is implemented as an SC_METHOD sensitive to the main clock. Implementing the stages as SC_METHOD works fine in a standalone architecture, with one processor and cache memory. However, multiprocessor execution will be impossible since the processor model will always own the SystemC execution context. In order to integrate the model in a SoC platform and to communicate with other SystemC IPs, we modify the stages to implement an SC_THREAD module and SystemC wait() function. This solution will not block the other IP modules from executing at the same clock cycle as the processor. A pseudo-code for the EX-stage module is shown in Figure 2.

To model the cycle-accurate pipeline correctly, the procedure is implemented as follows: each stage module executes in a while loop, and synchronizes with SystemC wait(). Only the first stage (IF) is sensitive to the main clock and to a synchronization signal (sync), while the others are sensitive to an input sync sent from the previous stage. When a new clock signal arrives, the IF-stage executes instruction i, and toggles the sync at its output. Then the ID-stage, which is sensitive to the sync from IF-stage, executes instruction i-1, and toggles its output sync. The same procedure repeats until WB-stage, which executes instruction i-4, and toggles the sync signal

With SC_METHOD

```
inline void
r3000_G_EX::r3000_G_EX::behavior()
{
 /* Get the instruction from the previous
stage register ID_EX */

 /* Decode the instruction */

 /* Call the generic instruction behavior */

 /* Call the format behavior */

 /* Call the instruction behavior */

 /* Send the instruction to the next stage
register EX_MEM */
}
```

With SC_THREAD

```
inline void
r3000_G_EX::r3000_G_EX::behavior()
{
  while (1) {

 /* Get the instruction from the previous
stage register ID_EX */

 /* wait (previous stage ID has finished); */

 /* Decode the instruction */

 /* Call the generic instruction behavior */

 /* Call the format behavior */

 /* Call the instruction behavior */

 /* Send the instruction to the next stage
register EX_MEM */

 /* notify the next stage MEM that the
results are ready */
   }
}
```

Figure 2. pseudo-code for the EX-stage module

which is connected back to the IF-stage. Finally, the IF-stage updates the internal pipeline registers and wait() for the next clock cycle. Note that the pipeline registers are double buffered for proper instruction execution in each stage. Figure 3 shows the modified R3000 cycle-accurate model that is generated by 'actsim'. This model can be integrated in a SoC simulator.

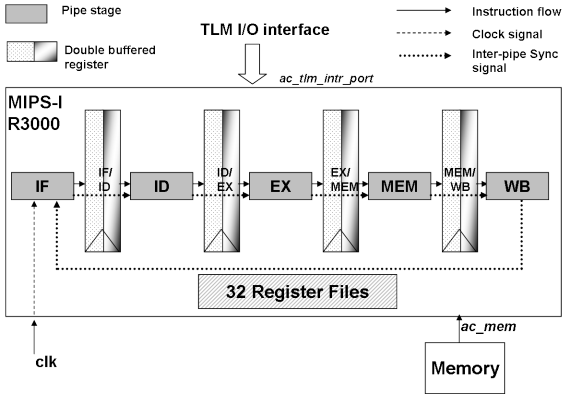

Figure 3. New R3000 cycle-accurate model for SoC simulator integration capabilities

The second modification done to the cycle-accurate simulator is the support of a TLM interface and an interruption mechanism. Since the functional simulator already implements the TLM interface, we reused the same code with some modifications to the interruption mechanism. According to the specifications in [17], the R3000 pipeline implements *precise exceptions* mechanism in order to avoid any type of pipeline anomalies. When an interrupt occurs, a 'trap' instruction is inserted in the IF-stage. The instructions in the pipeline finish their execution normally. When the 'trap' instruction reaches the WB-stage, it signals that the pipeline is now empty, and

that the execution of the interrupt service routine is allowed.

B. Modifications for variable pipeline depth

As we discussed in section IV-A, a SystemC module is generated for each pipeline stage by 'actsim'. Our objective is to duplicate an existing stage such as EX, into many stages, without creating a SystemC module for each of them and without recompiling the platform. The first EX-stage executes the real instruction behavior, and the latters are dummy EX-stages. They just forward the instruction data from one stage to the other, until it reaches the MEM-stage. In this way, an execution unit, such as FPU, can be simulated with variable execution time.

The traditional solution for adding a new pipeline stage requires modification of the *ac_pipe* variable in AC_ARCH file, regeneration then recompilation of the cycle-accurate model. This procedure should be repeated each time a new stage is added to the design. In addition, the r3000_isa.cpp should also be modified manually for each new architectural modification. Of course, this is not a handy process since it takes lot of development and debugging efforts.

Another alternative is to generate fixed maximum pipeline stages (i.e: 20 EX stages), then bypass the stages which does not take part in the simulation process. This static approach is not optimal, since it requires the generation of a SystemC module for each extra EX-stage, and the integration of a large complex ISA description file.

Our solution applies a dynamic approach with no simulator regeneration and recompilation. We denote by EXi the i^{th} extra EX stage. The AC_ARCH architecture description file remains the same as the 5-stage pipeline. We overload the processor constructor to take as input the desired number of extra stages. Then, the constructor instantiates the stages with their corresponding I/O signals and pipeline registers, and connects them dynamically to the other stages of the pipeline. For example, EX1 is connected to EX and EX2 stages, and EXn is connected to EXn-1 and MEM stage, where n is the total number of extra stages. The variable pipeline depth cycle-accurate model is shown in Figure 4.

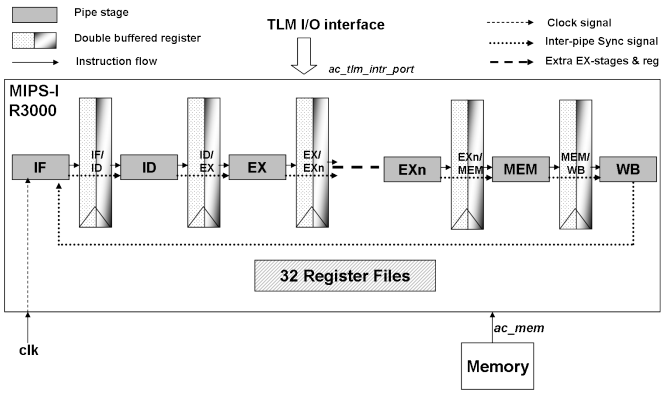

Figure 4. Variable pipeline depth R3000 cycle-accurate model for SoC simulator integration

The impact of the variable pipeline depth model on the execution process is elaborated in more details in the following subsections.

1) Pipeline anomalies: Implementing the variable pipeline depth model arises new data and branch hazards, which were previously resolved in the 5-stage pipeline.

The data hazards are the effect of data dependence between two instructions executed in the pipeline. In the 5-stage pipeline, 'register forwarding' solves this problem by bypassing values in late pipeline stages to earlier stages, hence no pipeline stalls will occur due to data hazards. However, when adding extra EXn stages, the data dependence check changes and a modified implementation of a 'register forwarding' technique is required. An instruction in the ID-stage checks the EX-stage first, then the extra EXn stages, and finally the MEM and WB stages. The search stops when the first instruction that holds the desired data for its operands is found. The data is forwarded back to the instruction in the ID-stage. The same procedure is repeated for an instruction in the EX-stage.

Pipeline stalls occur in the 5-stage pipeline when a branch instruction in the ID-stage requires a value from a further instruction in the pipeline, and the latter did not compute it yet. The maximum latency is 1 cycle. In the variable pipeline model, the stalls can occur on ID (branch hazards) and EX (data hazards) stages. This happens when an instruction in the EX-stage depends on an instruction (load instruction) that is still in the extra EXn stages, and that is waiting for memory access in MEM-stage. In this case, the pipeline should be stalled until the instruction in the extra EXn stages has finished its execution. In the 5-stage pipeline, this phenomenon does not occur, because the dependent instruction is already in the MEM-stage and register forwarding is implemented. The same reasoning is applied for branch instructions in the ID-stage.

In summary, we can see that in the variable pipeline depth model, the number of pipeline stalls varies according to the number of extra pipeline stages, as well as the instructions' dependency window in the program code.

2) ISA description file: Having a customized processor architecture necessitates a customized ISA description in r3000_isa.cpp file. The ISA description should be able to run properly for any number of pipeline stages. In [9], we see the implementation of the Type_R format behavior description for a 5-stage pipeline. Register forwarding is performed in the format behavior description (Type_R and Type_I), while branch hazards are checked in the ID stage of the 'branch instructions' behavior description (i.e: ac_behavior(beq), ac_behavior(jr)). Therefore, the ISA description modifications should be done for these 2 parts of the code in order to be generic. In Figure 5, we show the modifications in the EX-stage of the Type_R format behavior description, when checking the 'rs' operand register. The pseudo-code corresponds to the discussion we have conducted in section IV-B1. The same dependency is applied for the 'rt' operand register. The ID-

stage is implemented in a similar technique as the EX-stage. Notice the call of the pipeline stall function G_EX.stall(), when a dependency is found and cannot be resolved.

```
void ac_behavior(Type_R)
{

  SWITCH (stage)
  {
    ...
      CASE id_G_EX:

       /* check pipeline stall condition */

       FOR (i=0; i < (pipe_length - 5); i++)

            IF (a forwarding value exist in the extra EXn stages)
                BREAK;

       IF (a forwarding value exist) OR
          (a load instruction to one of the operand registers is in the MEM stage)
        {

            IF (!G_ID.is_stalled())

                 G_ID.stall();       /* stall the ID-stage */

            G_EX.stall();              /* stall the EX-stage */
        }

      /* Checking forwarding for the rs register */

       FOR (int i=0; i < (pipe_length - 5); i++) {

           IF (register writeback is found in the extra EXn stages) {
                 /* Get the writeback value */
                  BREAK;
            }
       }

       IF (no register writeback in the extra EXn stages) {

            IF (register writeback in MEM stage)
                  /* Get the writeback value */

            ELSE IF (register writeback in WB stage)
                  /* Get the writeback value */


            ELSE
                 /* Get the value from the ID-stage register */
       }
      ...
}
```

Figure 5. Modified Type_R format behavior description

As for the branch instructions, their ID-stage is modified so that it checks the extra EXn stages and stalls the pipeline if the dependence cannot be resolved. A pseudo-code for the jr instruction is shown in Figure 6.

Finally, a 'trap' instruction is added to the ISA description for proper pipeline interruption mechanism as discussed in Section IV-A. The application code does not have to be aware of this instruction, since the processor control part automatically inserts it in the IF-stage when an interruption occurs.

V. Pipeline statistics and debug

The performance evaluation of our cycle-accurate model necessitates the extraction of pipeline statistic values. Any degradation in the processor performance is mainly due to pipeline stalls. Those stalls arise from two types of sources: data dependencies (data and control hazards), and pipeline interlocks. The latter is due to memory access latencies when load/store instructions are in the MEM stage. In our model, we measure the total number of pipeline stalls due to data dependencies and pipeline interlocks. In addition, we sort them as a function of the number of stall cycles v/s the

```
//! Instruction jr behavior method.

void ac_behavior(jr)
{
    SWITCH (stage)
    {
      ...
      CASE id_G_ID:

              IF (no interruption in progress)

      /* Stalls the pipeline if the jump instruction depends on other instruction */

              FOR (i=0; i < (pipe_length - 5); i++) {

                    IF (a load instruction to one of the operand registers
                        is in one of the EXn stages) {

                        /* a dependency is found in one of the EXn stages */
                        BREAK;
                    }
               }

              IF (a dependency is found in one of the EXn stages) OR
                 (a dependency is found in the MEM stage) {

                       G_ID.stall();  /* stall the ID-stage */
              ELSE
                        /* Jump to the new address */
              }
     }

       ...
}
```

Figure 6. Modified ID stage for jr instruction behavior

number of occurences. In this way, we can know which type of instructions or series of instructions cause the most pipeline latencies, and dimension the processor pipeline accordingly. Furthermore, we are able to know which instructions missed the instruction or data caches by reading the occurence of the stalls for a specific latency. For example, if the cache and memory access needs 2 and 10 cycles respectively, then by reading the total number of stalls for these latencies we can deduce the percentage of cache misses. The code that measures those statistics is inserted in the processor model and the results are displayed automatically at the end of the program execution.

Moreover, we generate log files at the end of the program execution for all the processor pipeline activities. For every instruction in the ISA and for each pipeline stage, we integrate a debugging information that displays the currently existing instruction, stage, operands, and pipeline status (whether stalled or not). This is extremely useful for visualizing the program execution in the pipeline and solving ISA problems. In order to set the debug option, we include -DDEBUG_PROC_PIPE and -DDEBUG_PROC_REG_RB options in the Makefile before generating the model.

VI. Results

The MIPS-I cycle-accurate ISS model simulates the processor performance almost as accurate as the RTL model. This advantage comes at the expense of the simulation speed. For this reason, we will conduct 2 experiments that investigate the different ArchC models simulation speed, as well as their pipeline accuracy levels.

Our simulations were performed on an Intel(R) Core(TM) 2 Quad CPU running at 2.83 GHz with 10 GB of RAM. They are launched from a python script, which generates 4 executing threads, each corresponding to one simulation,

Program	# of instructions	Functional	Cycle-accurate					
			SC_METHOD	SC_THREAD				
		Perf. (KIPS)	5-stages	5-stages	6-stages	7-stages	8-stages	9-stages
			Perf. (KIPS)	Perf. (KIPS)	Perf. (KIPS)	Perf. (KIPS)	Perf. (KIPS)	Perf. (KIPS)
bitcount	45593673	12061.82	121.74	41.46	28.26	29.33	23.62	17.78
qsort	14412622	11911.26	124.09	42.73	29.68	21.22	20.06	15.86
susan(corners)	3458871	11529.57	125.19	43.15	35.08	23.38	18.89	18.8
susan(edges)	6887632	11875.23	124.55	43.53	29.41	24.1	21.31	17.54
susan(smoothing)	35320188	10935.04	126.56	44.26	34.04	24.86	20.33	13.55
jpeg_encoder	29474822	11789.93	122.51	35.27	31.9	26.3	20.98	13.79
jpeg_decoder	8697310	11295.21	132.04	38.29	36.11	22.46	21.48	16.75
stringsearch	279724	11811.45	123.23	34.07	25.22	19.78	21.05	14.13
rijndael_encoder	33715297	11546.33	131.06	34.74	30.68	24.78	22.56	17.98
rijndael_decoder	34684743	11757.54	132.92	36.12	37.91	24.8	21.56	17.39
sha	13036286	11639.54	133.8	38.81	38.63	32.08	25.93	17.4
adpcm_encoder	34628835	12501.38	113.77	39.5	32.6	26.71	19.65	14.22
adpcm_decoder	27256673	12114.08	114.76	39.65	32.56	26.8	22.38	13.71
adpcm_timing	300730080	11933.73	114.74	32.17	26.22	21.64	22.61	15.89
CRC32	31643638	11896.1	128.98	43.56	29.69	19.4	17.96	8.84
gsm_encoder	32663572	11921.01	130.64	38.01	37.88	25.1	24.91	14.13
gsm_decoder	9614567	11869.83	135.38	37.22	34.35	22.35	18.09	12.3

Table I
MIBENCH BENCHMARK SUITE ON MIPS-I FUNCTIONAL MODEL, MIPS-I R3000 CYCLE-ACCURATE MODEL WITH SC_METHOD, SC_THREAD, AND VARIABLE PIPELINE DEPTH

and affines them to one CPU core. We simulate most of the MiBench embedded benchmark suite [2] that is already cross-compiled to MIPS-I architecture and is available on [18]. For all the experiments, the models are simulated with an infinite cache. Therefore, there are no performance degradations due to 'pipeline interlocks' or long latencies memory operations. The only source of performance degradation is the pipeline stall due to control/data hazards.

In the first experiment, we show the differences in the simulation speed between a MIPS-I functional model, MIPS-I R3000 cycle-accurate model with SC_METHOD stages and MIPS-I R3000 cycle-accurate model with SC_THREAD stages. The latter is simulated for different pipeline lengths which correspond to the extra EXn stages (n=0,1,2,3,4). The results are shown in Table I.

As expected, the simulation speed drops to an approximate ratio of 100 between the functional and cycle-accurate SC_METHOD models, and 3.5 between the cycle-accurate SC_METHOD and SC_THREAD models. This difference between the SC_METHOD and SC_THREAD models is due to lot of context switches in the SystemC kernel for the SC_THREAD modules. However, this solution is important for multiprocessor and SoC simulations as previously discussed in Section IV-A. We notice also that the simulation speed drop varies by adding an extra EX stage, since the number and penalty cycles of pipeline stalls changes from one application to the other.

According to [19], the simulation speed of SimpleScalar and MC-Sim (based on SESC simulator) are around 150 KIPS and 32 KIPS respectively. The former is a standalone architecture while the latter is integrated in an MPSoC design. We can reach similar values with the SC_METHOD model (standalone architecture) and SC_THREAD model (MPSoC).

The second experiment shows the efficiency of the cycle-accurate SC_THREAD model by calculating the CPI (clock per instruction) for each pipeline length for the MiBench programs. By using the pipeline statistics information, we are able to measure the penalty cycles of the different pipeline stalls, and categorize them accordingly. In Figure 7, we show the results for pipeline configurations with no extra EXn stages (5-stage pipeline) and with 3 extra EXn stages (8-stage pipeline). For the 5-stage pipeline, all the stalls have a 1 cycle penalty because we implement the 'register forwarding' technique. We see also that the CPI varies with respect to the application code, which was not possible to detect with the functional ISS model. As for the 8-stage pipeline, the CPI increases for all the applications, since the extra EXn stages introduce new dependencies. In addition, the stalls penalty cycles are more severe and can reach 4 cycles. In recent processor architectures such as the Pentium (20-stage pipeline), the penalties due to pipeline dependencies in a large pipeline depth can be resolved with an out-of-order processing. These penalties would be more severe if the memory model is simulated accurately with a latency access.

VII. CONCLUSION AND FUTURE WORKS

This paper presented a cycle-accurate ISS model based on ArchC 2.0 and MIPS-I R3000 architecture, with a parameterizable number of EX stages. The ISA description is independent from the number of pipeline stages. The techniques that have been developed, both for standalone architecture (using SC_METHOD) and multiprocessor architecture (using SC_THREAD) are efficient. With a reasonable development effort, they allow to build fast cycle-accurate instruction set simulators that will be used to evaluate performance of various multi-thread and multi-core architectures for embedded systems.

The model is validated by executing the MiBench embedded

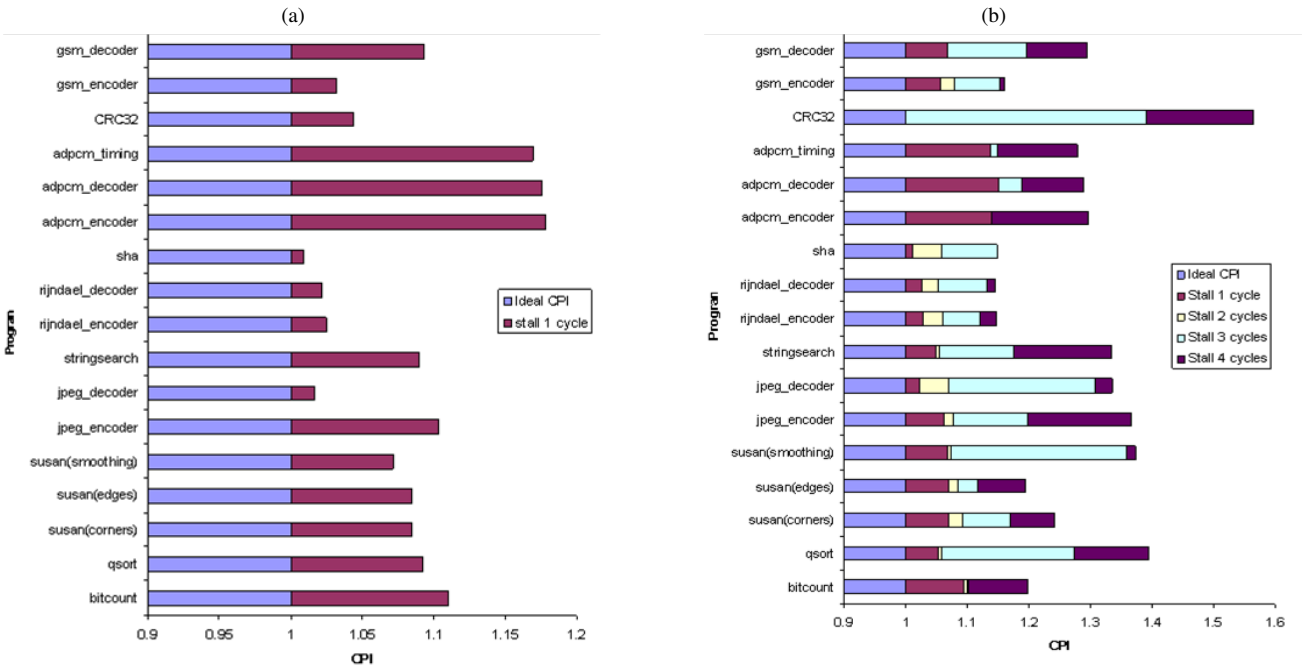

Figure 7. CPI performance for MiBench suite for different number of extra EXn stages. (a) EXn=0 (b) EXn=3

benchmark suite on different pipeline configurations. The applications' CPI performances are measured, while differentiating the type of stalls as well as their penalty cycles. Results show the ability to dimension the processor architecture according to the characteristics of each application code.

In our first implementation, we allow the EX-stage to have a variable length. However, future enhancements of the model will allow the parameterization of any pipeline stage and adapt the ISA accordingly. Also, possible extensions to the ArchC language would be the support of different processor components and out-of-order architectures.

REFERENCES

[1] A. A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. Elsevier, 2005.

[2] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec. 2001.

[3] Rainer Leupers and Peter Marwedel. Retargetable code generation based on structural processor descriptions. In *In Design Automation for Embedded Systems*, pages 1–36. Kluwer Academic Publishers, 1998.

[4] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[5] G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: An instruction set description language for retargetability. In *Design Automation Conference, 1997. Proceedings of the 34th*, pages 299–302, Jun 1997.

[6] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM.

[7] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 47–56, New York, NY, USA, 2004. ACM.

[8] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa—machine description language for cycle-accurate models of programmable dsp architectures. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 933–938, New York, NY, USA, 1999. ACM.

[9] M. Bartholomeu G. Araujo C. Araujo R. Azevedo, S. Rigo and E. Barros. The ArchC Architecture Description Language and Tools. *Parallel Programming*, 33(5):453–484, 2005.

[10] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. Archc: a systemc-based architecture description language. In *Proc. 16th Symposium on Computer Architecture and High Performance Computing SBAC-PAD 2004*, pages 66–73, 2004.

[11] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto. Resp: A non-intrusive transaction-level reflective mpsoc simulation platform for design space exploration. In *Proc. Asia and South Pacific Design Automation Conference ASPDAC 2008*, pages 673–678, 2008.

[12] M. R. de Schultz, A. K. I. Mendonca, F. G. Carvalho, O. J. V. Furtado, and L. C. V. Santos. Automatically-retargetable model-driven tools for embedded code inspection in socs. In *Proc. 50th Midwest Symposium on Circuits and Systems MWSCAS 2007*, pages 245–248, 5–8 Aug. 2007.

[13] N. Kavvadias and S. Nikolaidis. Elimination of overhead operations in complex loop structures for embedded microprocessors. 57(2):200–214, Feb. 2008.

[14] Open SystemC Initiative:. http://www.systemc.org.

[15] C. Araujo, M. Gomes, E. Barros, S. Rigo, R. Azevedo, and G. Araujo. Platform designer: An approach for modeling multiprocessor platforms based on systemc. *Design Automation for Embedded Systems*, 10(4):253–283, 2005.

[16] Sandro Rigo, Marcio Juliato, Rodolfo Azevedo, Guido Araújo, and Paulo Centoducatte. Teaching computer architecture using an architecture description language. In *WCAE '04: Proceedings of the 2004 workshop on Computer architecture education*, page 6, New York, NY, USA, 2004. ACM.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[18] ArchC official website. http://archc.sourceforge.net/.

[19] J. Cong, K. Gururaj, G. Han, A. Kaplan, M. Naik, and G. Reinman. Mcsim: An efficient simulation tool for mpsoc designs. In *Proc. IEEE/ACM International Conference on Computer-Aided Design ICCAD 2008*, pages 364–371, 2008.