A TLM-based Multithreaded Instruction Set Simulator for MPSoC Simulation Environment

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;

***Abstract*—With the increase in the design complexity of MPSoC architectures, flexible and accurate processor simulators become a necessity for exploring the vast design space solutions. In this paper, we present a flexible multithreaded ISS model based on a modular cycle-accurate modeling technique. The model is scalable for n hardware threads and implements two thread scheduling algorithms: interleaved and blocked multithreading. Thanks to its TLM interfaces, the ISS can be easily integrated in any MPSoC design based on SystemC. Its performance and capabilities are demonstrated by running two applications of MiBench embedded benchmark suite, showing the advantage of multithreaded processors through pipeline statistics extraction.**

keywords: ISS, multithreaded processors, cycle-accurate, SystemC, TLM, System-on-Chip, ADL, Design Space Exploration

I. INTRODUCTION

Traditional high-performance superscalar processors implement several architectural enhancement techniques such as out-of-order execution, branch prediction, and speculation, in order to exploit the instruction-level parallelism (ILP) of a sequential program. However, due to the limits of ILP [1], a more coarse-grained solution consists of exploiting the parallelism at the thread level (TLP), where multiple threads can be executed in parallel on multicore processors or concurrently on hardware multithreaded processors. A hardware multithreaded processor [2] provides the hardware resources and mechanisms to execute several virtual threads on one processor core in order to increase its pipeline utilization, hence the application throughput. The virtual threads compete for the shared resources and tolerate pipeline stalls due to long latency events, such as cache misses. These events can stall the pipeline up to 75% of its execution time [3]. Multithreading can be applied on single-issue scalar architectures such as interleaved multithreading (IMT) and blocked multithreading (BMT) processors, and wide-issue superscalar architectures such as simultaneous multithreading (SMT) processors.

The emergence of new multithreaded embedded applications for network, telecom, automotive, digital television and multimedia applications, has fueled the demand for architectures that are more transistor and energy efficient. Thus, hardware designers are showing interest in scalar multithreaded RISCs for future integration in MPSoC architectures [4]. For instance, the MIPS34K [5] and Infineon TriCore 2 [6] are two examples of commercial multithreaded IP cores for embedded systems.

At the early stage of MPSoC development, a system designer needs to perform rapid design space exploration, preferably in a unified development environment such as SystemC [7], in order to find the best trade-off solution. Choosing the best multithreaded processor among several architectures necessitates the evaluation of many different features such as the number of hardware threads, the pipeline structure, the ISA description, and the register files. An Instruction Set Simulator (ISS) mimics the behavior of a processor by executing target processor instructions while running on a host computer. An ISS must be parameterizable, fast, accurate, and easily integrable in the MPSoC simulation environment. In addition, a multithreaded ISS must be scalable with the number of hardware threads and must have a short development and validation time.

In a previous work [8], we have developed a cycle-accurate monothreaded ISS based on TLM interfaces, which can be integrated in a SystemC multiprocessor environment. In this paper, we expand the cycle-accurate ISS model in order to provide a modular cycle-accurate multithreaded ISS ready to be integrated in an MPSoC simulation environment. The modular cycle-accurate model encapsulates n cycle-accurate ISS and a *scheduler* into one module, which mimics the behavior of a scalar multithreaded RISC. Since it is based on cycle-accurate simulators, the multithreaded ISS is able to catch all the pipeline stalls that exist in a real processor architecture. Currently, we support 2 types of multithreaded processors: IMT and BMT. However, other types of scalar multithreaded processors can be easily implemented as it is described in this work.

This paper is organized as follows: Section II discusses related works on different types of ADLs and simulators. Section III introduces briefly the monothreaded MIPS-I cycle-accurate ISS model, which is instantiated n times in the multithreaded model. Then, Section IV gives an overview of the multithreaded cycle-accurate ISS based on a modular modeling technique. It describes the interleaved and blocked multithreading ISS models. A case study scenario in which

the two models are validated using 2 MiBench [9] applications (qsort, CRC32) is provided in section V. Finally, section VI concludes the paper by discussing the present results along with future works.

II. Related Work

Lot of works has been published before on single-processor, multiprocessor and full-system simulators. In [10], the authors illustrate a wide range of simulators, mainly targeting general-purpose computing. In a more recent work [11], Jason Cong et al presented an interesting classification of MPSoC simulators. In this section, we will mainly concentrate on ADLs and Instruction Set Simulators (ISS) based on SystemC description language, which are relevant to our research.

A SystemC/ISS co-simulation environment provides design flexibility, by being able to experiment with different types and numbers of processor architectures at the early design stages. This advantage has led researchers [12] to provide SystemC wrappers for traditional standalone ISS such as SimpleScalar [13]. Other works [14], [15] used the same technique for integrating a non-native SystemC ISS into a SystemC/ISS co-simulation environment. However, the main drawback of the SystemC wrappers approach is the slow simulation speed (order of few KIPS) with respect to a standalone ISS (order of hundred KIPS to MIPS).

On the other hand, standalone multithreaded simulators exist in the literature, mainly targerting SMT type of processors. For example, SSMT [16] and M-SIM [17] are SMT extensions on top of SimpleScalar. Other simulators, such as SESC [18] and Sam CMT Simulator kit [19], support the simulation of chip multithreaded (CMT) processors. Despite of their flexibility and parameters variability, these full-system simulators are standalone and require SystemC wrappers with TLM interfaces to be interfaced with other SystemC components.

In most cases, the ISS is generated by an Architecture Description Language (ADL) at a specific level of abstraction. ADLs' modelization levels are classified into three categories: structural/cycle-accurate such as MIMOLA [20], behavioral/functional such as nML [21], and mixed such as LISA [22]. However, these ADLs do not generate an ISS in SystemC language. A recent type of processor description language called ArchC [23] is gaining special attention from the research communities [24], [25]. ArchC 2.0 is an open-source 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. 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 automatically generated according to the architecture resource description file.

To the authors' knowledge, there does not exist in the literature any IP-based multithreaded ISS in SystemC with TLM-based interfaces for MPSoC design space exploration, which is the scope of this work.

III. Monothreaded MIPS-I R3000 cycle-accurate ISS 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 version described in [26]. A cycle-accurate ISS model is developed based on ArchC 2.0 and MIPS-I R3000 architecture, with a parameterizable number of EX stages [8]. In fact, the pipeline depth is one important parameter for processor sizing. A deeper pipeline allows a higher clock frequency, but leads to increased load-use latencies, branch latencies and mispredicted branch penalties. In any case, multi-cycle instructions, such as integer multiplication/division and all the floating-point instructions, are mandatory. The variable pipeline depth model of the cycle-accurate MIPS-I ISS is shown in Figure 1.

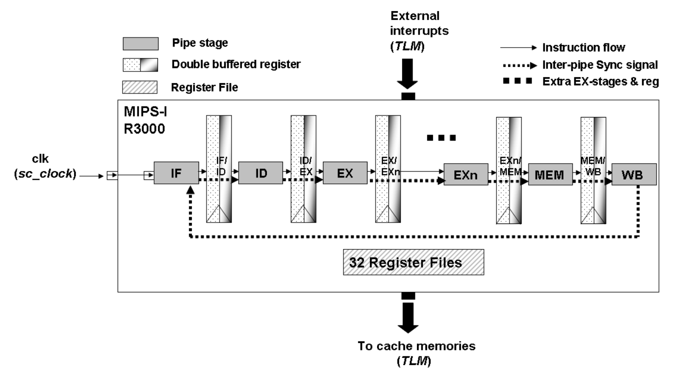

Figure 1. Block diagram of the variable pipeline depth model within the cycle-accurate MIPS-I R3000 ISS

The ISS SystemC module implements 2 TLM I/O interfaces: the first one receives interrupts from external sources such as a controller (*sc_export*), and the second one sends memory access requests to the memory (*sc_port*). The R3000 pipeline implements *precise exceptions* mechanism in order to avoid any type of pipeline anomalies [26]. So whenever an external interrupt occurs, the R3000 pipeline is flushed. Then, the appropriate interrupt service routine is called depending on the interrupt type. For instance, we support 3 types of TLM interrupts: *start* a new task, *preempt* the current task with a new task, and *stop* the current task. The TLM interrupt protocol is a modified ArchC TLM protocol [23].

Since the ISS is cycle-accurate, it has also a sc_clock input signal. Internally, each pipeline stage module is a SystemC SC_THREAD module, synchronized with a SystemC wait() function. Only the first stage (IF) is sensitive to the main clock and to a synchronization *sync* signal, while the others are sensitive to an input *sync* signal sent from the previous stage. When a new clock signal arrives, the IF-stage processes instruction i, and toggles the *sync* signal at its output. Then,

the ID-stage, which is sensitive to the *sync* signal from IF-stage, processes instruction i-1, and toggles its output *sync* signal. The same procedure repeats until the WB-stage, which processes instruction i-n-4, where n is the number of extra EX-stages. Then, it toggles the *sync* signal 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 processing in each stage, and the extra EX-stages are dummy stages.

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 long memory access latencies when load/store instructions are in the MEM stage. In our model, we can measure the total number of pipeline stalls due to data dependencies and pipeline interlocks.

This cycle-accurate ISS can only execute one virtual thread at a time. The next section describes the development a multithreaded ISS, which is able to execute multiple threads at a time.

IV. MULTITHREADED CYCLE-ACCURATE ISS MODEL

The multithreaded ISS is designed to be integrated in a typical processor system environment, based on SystemC language. It keeps the same TLM I/O interfaces as the monothreaded ISS described in section III, in order to look as one ISS/processor to the external world. We use a typical processor system environment described in Figure 2. It implements a 2-level memory hierarchy with L1$ and L2$ shared by all the virtual threads. The processor-memory architecture is based on a Harvard architecture with separate L1 I$ and D$ busses. The bus multiplexing information is added to the TLM protocol (based on the ArchC TLM protocol). This information is set by the cycle-accurate ISS, which redirects a TLM packet to the I$ in case of an instruction fetch, or D$ in case of a memory read/write. In addition, we implemented a *controller/thread allocator* module, which dispatches active threads to the processor based on a scheduling algorithm (i.e. round-robin). All the I/O TLM interfaces are blocking, which means that the sender is blocked until it gets a response from the receiver. For example, when the processor issues a TLM memory request to the L1 I$ and a cache miss occurs, the L1 I$ re-issues a TLM memory request to the L2$ and the response is sent back to the processor. During this time, the processor is blocked and cannot execute other requests. In section IV-B, we show how we will overcome this issue using the BMT ISS model.

The multithreaded ISS uses a modular cycle-accurate technique to mimic the behavior of a scalar multithreaded RISC. It encapsulates *n* pre-validated cycle-accurate ISS for the MIPS-I R3000 (see section III), each corresponding to one virtual context. It receives TLM interrupt requests from an external module such as a hardware controller, and sends TLM memory access requests to the caches. Internally, a *scheduler* module synchronizes and schedules all the memory access requests of the *n* ISS. Figure 3 shows the internal structure of the multithreaded ISS model, denoted by PE_MT. Each R3000 ISS[i] in PE_MT simulates only the pipeline stages, which are described previously in Figure 1. The R3000 ISS[i] is generated automatically in ArchC using 'actsim' tool as described in [8], while the other block modules (scheduler, TLM demultiplexer) are developed in SystemC.

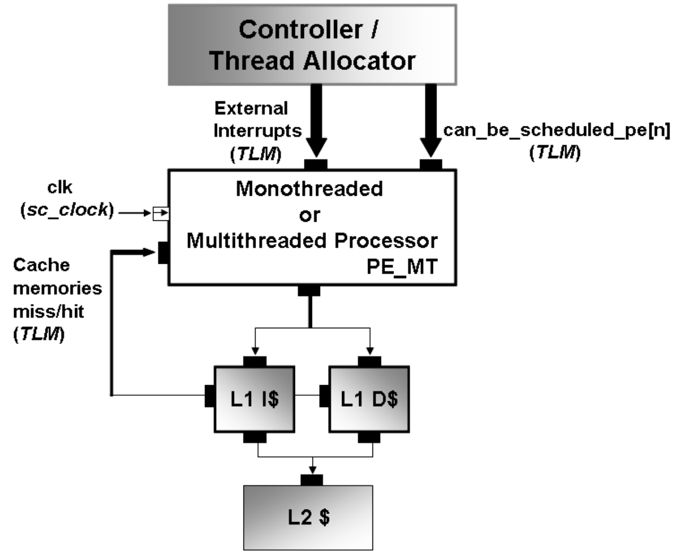

Figure 2. Processor system model

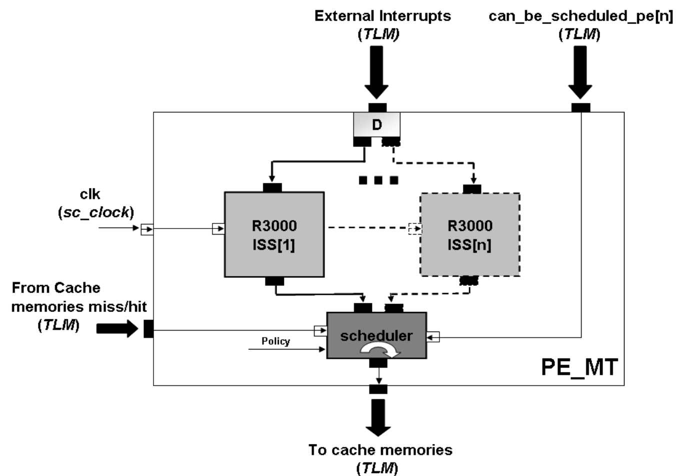

Figure 3. Multithreaded ISS model

For the controller, the PE_MT looks as *n* virtual processors. Each internal ISS is a virtual thread; therefore it has a unique id (vt_id). A vt_id parameter is added to the TLM protocol, so that every incoming and outcoming TLM packet can be tracked in the platform. All the external interrupts are input to a 1-to-n TLM demultiplexer (labeled as D in Figure 3). It checks the vt_id of the TLM packet and then forwards it to the

corresponding ISS. Then, the ISS handles the request, updates its internal state and executes the corresponding task. It generates two types of TLM memory requests: an instruction fetch from the IF-stage and a data memory access from the MEM-stage. The *scheduler* module receives TLM memory requests from the n ISS. It synchronizes and schedules the packets according to a pre-defined scheduling policy implemented as an FSM diagram. Then, it selects one of these packets at a time and transfers it to the cache memories. In order to facilitate the scheduling decisions of the *scheduler*, we provide 2 types of information as input to the multithreaded module:

1) The scheduling status (active/idle) of each ISS, which comes from the external controller using can_be_scheduled_pe[n] input (*sc_export TLM*), where n designates the virtual thread id.
2) The caches hit/miss input (*sc_export TLM*), which inform the scheduler the status of each memory access request

In this work, we implement 2 multithreading scheduling techniques: IMT and BMT. Each one has its own FSM diagram implemented in the *scheduler*. The IMT is implemented as a Mealy FSM and BMT as a Moore FSM. Therefore, to add a new multithreading technique, the designer just have to embed the FSM diagram code in the *scheduler* without modifying the other components.

Note that the *scheduler* module is not clocked and is only synchronized by SystemC events. This is important when a functional ISS (not clocked) is used instead of a cycle-accurate ISS, which makes the *scheduler* more general.

A. Interleaved multithreading ISS

An IMT processor executes an instruction from one active thread at a time in a round-robin way. Therefore, in any 2 consecutive pipeline stages, there is an instruction from a different virtual thread. For example, in the R3000 5-stage pipeline architecture, 2 virtual threads are enough for eliminating pipeline stalls due to data dependencies. However, if one thread is waiting for a long latency memory access, then the whole pipeline is stalled.

To model this behavior using *n* separate ISS, the scheduler FSM should allow the execution of one ISS pipeline until completion, and then switch to another active ISS pipeline in zero cycles. During the pipeline execution cycle, it generates a maximum of 2 TLM memory requests, one from IF-stage and one from MEM-stage. The FSM switches the thread execution whenever an ISS pipeline is fully processed. Therefore, we differentiate between an IF-stage and a MEM-stage TLM packet by adding a parameter to the TLM protocol. The FSM for the IMT model with 2 virtual threads, shown in Figure 4, is implemented as a Mealy FSM.

Since each ISS is cycle-accurate, small latency pipeline stalls due to data dependencies are captured by the ISS itself. As for the long latency pipeline stalls (i.e. cache miss), they are modeled intuitively by the TLM interface blocking mechanism.

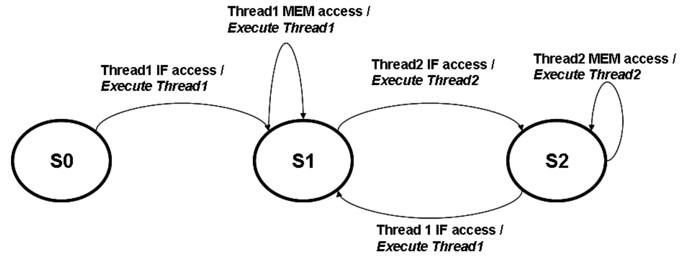

Figure 4. Interleaved multithreading scheduler FSM (Mealy machine)

The sequential thread program execution on each ISS does not reflect the actual behavior of the IMT pipeline. In fact, the execution speed of each thread should be divided by *n* and the pipeline stalls due to data dependencies should be eliminated. This is done by inserting *n-1* "dummy nop" instructions after each fetched instruction. The "dummy nop" instruction does not access the memory, thus does not generate an IF-stage TLM request and is transparent to the *scheduler*. This requires a slight modification to the IF-stage code of the original MIPS-I R3000 model. A pipeline execution model of 2 virtual threads is shown in Figure 5.

	PE1	PE2	PE_MT
Cycle i			
IF:	add	nop (dummy)	**add (pe1)**
ID:	nop (dummy)	sub	**sub (pe2)**
EX:	beq	nop (dummy)	**beq (pe1)**
MEM:	nop (dummy)	j	**j (pe2)**
WB:	sub	nop (dummy)	**sub (pe1)**
Cycle i+1			
IF:	nop (dummy)	add	**add (pe2)**
ID:	add	nop (dummy)	**add (pe1)**
EX:	nop (dummy)	sub	**sub (pe2)**
MEM:	beq	nop (dummy)	**beq (pe1)**
WB:	nop (dummy)	j	**j (pe2)**

Figure 5. Interleaved multithreading pipeline representation

As we can notice, by overlapping the pipeline stages of all the ISS ("dummy nop" are transparent), we get the pipeline behavior of a scalar IMT processor.

Finally, the *scheduler* should keep track of the scheduling status of each virtual thread using can_be_scheduled_pe[n] input signals from the controller. If one of the threads is scheduled/descheduled, then the *scheduler* informs the other ISS to adjust the number of "dummy nop" instructions.

B. Blocked multithreading ISS

A BMT processor executes one thread as on a mono-processor, and switches to another thread whenever a long latency event occurs, such as a cache miss. Thus, small latency pipeline stalls such as pipeline dependencies are not tolerated by this model. Therefore, a thread status is defined as:

1) ACTIVE: if Thread[i] is scheduled and executing properly without long latency events
2) NOT ACTIVE: if Thread[i] is not scheduled by the controller or has a long latency event such as a cache miss and TLB miss or is stalled on data synchronization with another Thread[j] (future work).

The *scheduler* FSM requires external signals from the caches (Cache memories miss/hit signals shown in Figure 3) in order to perform its decision. In our work, we implement a "greedy" BMT protocol, where one main thread (R3000 ISS[1]) has a higher priority than the others (R3000 ISS[2] to R3000 ISS[n]), thus its execution speed is not altered. This scenario considers that the low priority threads are helper threads. However, if there are not enough memory stall latencies, the "greedy" protocol may cause starvation to some helper threads. The FSM diagram for 2 virtual threads, shown in Figure 6, is implemented as a Moore FSM.

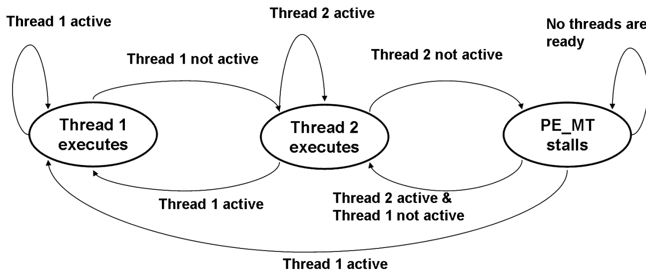

Figure 6. Blocked multithreading scheduler FSM using greedy protocol (Moore machine)

Initially, Thread1 executes as long as there is a cache hit or small latency event. Whenever there is a miss, Thread2 starts the execution and fills the stalling slot cycles of Thread1. When Thread1's data is returned, it resumes the execution. Otherwise, Thread2 continues the execution until there is a miss. Then the whole processor is stalled and waits for one of the threads' returned data in order to resume the execution, with a higher priority to Thread1 in case of a simultaneous response.

On the contrary to the IMT model, the BMT model does not require any changes to the monothreaded ISS, such as "dummy nop" insertions and memory access packet distinction. The latter implies that the BMT model is sensitive to a long latency event, whether it comes from an IF-stage or MEM-stage packet.

V. CASE STUDY

In this part, we provide a case study scenario in order to validate the functionality of the multithreaded models in an embedded SoC platform and show the different generated statistics that will help the designers to perform design space exploration. However, this part is not designated to be a DSE for the overall platform.

We will use two applications from the MiBench embedded benchmark suite: qsort and CRC32. The two applications are chosen due to their simplicity and ease of portability to our framework. Each application executes in one thread context. The two applications run sequentially on the monothreaded processor and concurrently on the multithreaded processors. The application input data are adjusted in order to have a reasonable simulation time. The application binaries are generated by the MIPS gcc 4.2.3 cross-compiler.

In order to validate the models, we use the SESAM framework [27] to build the typical processor system environment previously explained in section IV. All the modules, except the caches (modified ArchC cache models), are designed by our laboratory. They are parameterizable and ready to be integrated in an MPSoC design. A Python script is used to automatically generate several simulation instances by varying different architecture parameters. An Excel macro imports the generated statistics and plots the results in graphs, similar to those shown in this paper.

In an embedded SoC, the transistor efficiency of any design choice is critical. Therefore, the number of virtual contexts should be specified in the multithreaded processor. We synthesized the MIPS-I R2000 (similar to R3000 architecture) processor [28] with a 40 nm TSMC technology. The processor die area without the memories is 16567 um^2. The repartition of the different processor's components areas is shown in Figure 7. Based on these values, we estimate the total die area increase for a multithreaded processor with 2 virtual contexts by doubling the register file and PC, adding 1-bit inter-stage register and miscellaneous control components. The estimated die area is 24069 um^2, which is approximately **45%** more than the monoprocessor area. Therefore, there is a diminishing return advantage of implementing a larger embedded multithreaded processor. Hence, we choose 2 virtual contexts.

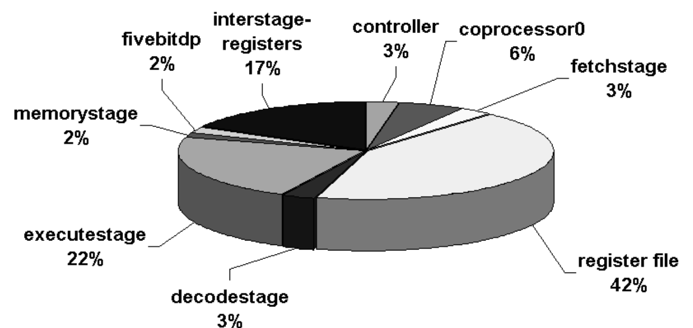

Figure 7. MIPS synthesis

In this experiment, we vary 3 platform parameters for a better architecture exploration. First, the processor type can be chosen to be *monothreaded*, *IMT* or *BMT*. Second, the processor pipeline depth can be set to *5*, *6* or *7* stages. The variable number of EX-stages in the pipeline models an execution unit with more than 1-cycle latency such as a floating point unit. Third, the L2$ memory access request due to a cache miss can take *3*, *5*, or *10* clock cycles. This allows

us to model different on-chip memory technologies. On the other hand, the L1 cache configurations are fixed for a typical embedded context. The cache size is 16 KB, the block size is 8B, the word size is 32B, and the set-associativity is 4. We assume that a cache hit takes 1 clock cycle.

In Figure 8, we fix the pipeline depth to 5 and vary the L2 memory access latency in order to observe the behavior of each processor pipeline. We denote by effective IPC the time spent by the processor executing effective instructions.

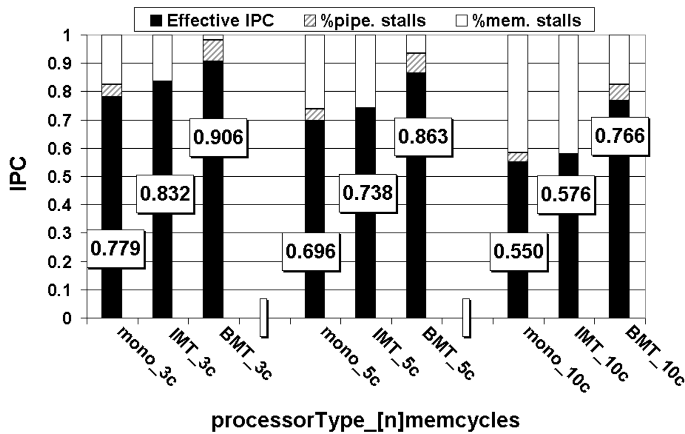

Figure 8. Effective IPC v/s memory access latency. The x-axis represents the processor type and the number of L2$ memory access cycles

The maximum IPC that a scalar processor can reach is 1. In order to understand the sources of IPC degradation, we decompose the maximum IPC into 3 components: effective IPC, IPC lost due to pipeline stalls and IPC lost due to memory stalls. In Figure 8, we notice that the BMT processor tolerates better the long memory stall latencies. On the other hand, the IMT processor tolerates the pipeline dependency stalls. By increasing the L2 memory access latency, the effective IPC drops for all the processor configurations due to the increase of memory stalls. Since the monothreaded configuration tells us that the largest percentage of processor stalls are due to memory stalls, then the effective IPC of the BMT overcomes that of the IMT for all memory latency values.

In another experiment, we vary the number of pipeline stages and the memory latencies, then we compare the effective IPC of the 3 processors configurations in Figure 9. By increasing the processor pipeline depth, the pipeline dependency stalls have a bigger impact. This phenomenon is shown in Figure 9(a) for the monothreaded configuration: the effective IPC drops between 23% and 30% by varying the pipeline depth from 5 to 7 for the different memory access cycles. In Figure 9(b), we see that the IMT processor is more tolerant to these pipeline dependency stalls. Its effective IPC drops by a maximum of 6% when increasing the pipeline depth. However, Figure 9(c) shows that the effective IPC of the BMT processor is more sensitive to the pipeline depth, which can lead to 35% of performance drop. We confirm that the BMT processor can better tolerate long latency stalls. For instance, BMT scores a maximum of 18% performance drop, whereas the IMT can reach up to 45%.

Finally, in Figure 10, we measure the speedup of each multithreaded model with respect to the monothreaded processor with the same pipeline stage.

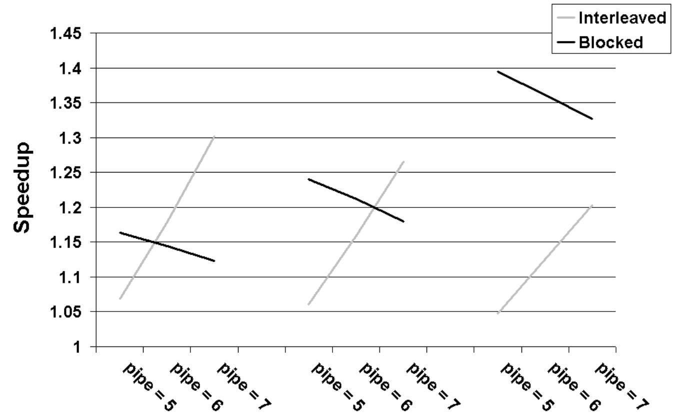

Figure 10. Speedup with respect to a monothreaded processor for the same pipeline depth

The speedup values tell us that the choice of the multithreading technique depends on the original processor pipeline architecture and the on-chip memory technology used. Therefore, having a flexible cycle-accurate ISS allows us to choose the best multithreading technique for a specific MPSoC architecture.

The simulation speed of the multithreaded ISS, which is the aggregate sum of the *n* ISS, does not degrade in performance compared to that of the monothreaded ISS [8]. We can reach an aggregate simulation speed up to *40 KIPS*. To our knowledge, SESC is the closest comparable simulator to our design and can reach 32 KIPS when integrated in an MPSoC design [11]. It is worth to note that the presented results are pretty well known in the literature, and they are shown only to validate the correctness of our multithreaded ISS model.

VI. Conclusion

This paper presented a multithreaded ISS model based on a modular cycle-accurate modeling technique. It encapsulates several MIPS-I R3000 cycle-accurate ISS and synchronizes their memory accesses with a scheduler. The scheduler implements a specific thread scheduling algorithm represented in an FSM diagram. In this paper, we have developed the FSM of the IMT and BMT processors. New multithreaded models can be easily built with less than one-day man work by modifying the code of the *scheduler* FSM. The model allows to build cycle-accurate scalar multithreaded ISS that will be used to evaluate performance of various multithreaded and multicore architectures for embedded systems.

The model is validated by executing two applications of the MiBench embedded benchmark suite (qsort, CRC32), while varying the memory access latencies and the pipeline

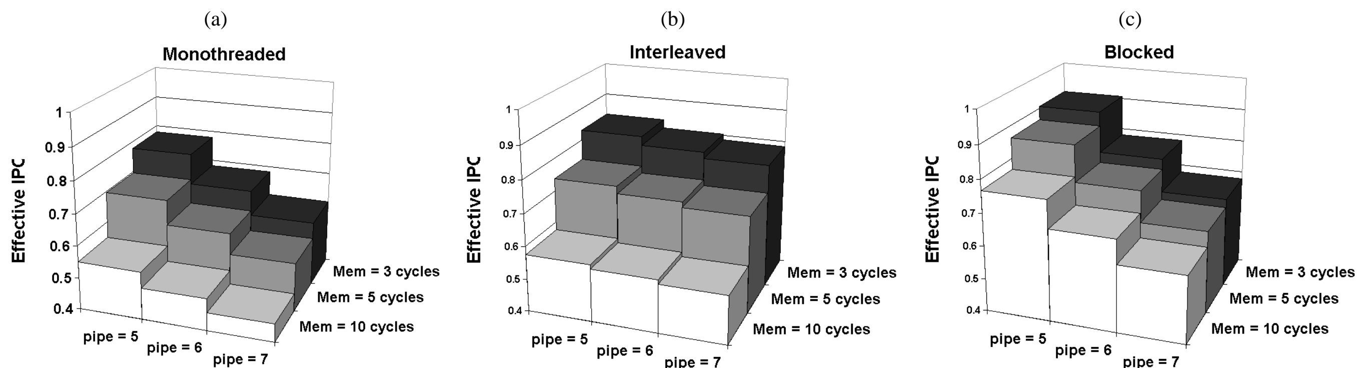

Figure 9. Effective IPC v/s memory access latency v/s pipeline depth for: (a) monothreaded (b) IMT (c) BMT processors

depth. The extracted pipeline statistics showed that an optimal multithreading technique depends on the processor pipeline architecture and on the on-chip memory technology used in the SoC platform. The interleaved multithreading favors deeper pipeline depths, while blocked multithreading favors long memory access cycles.

For future enhancements, we aim to implement a superscalar SMT model and test different thread scheduling algorithms. In addition, we aim to design an automated tool that generates the *scheduler* FSM skeleton to reduce the development time. For future work, we want to validate the accuracy of our model against a real hardware implementation, and to conduct a DSE study of a MPSoC platform consisting of multiple multithreaded cores and executing real multithreaded benchmarks.

REFERENCES

[1] D.W. Wall. Limits of instruction-level parallelism. In *ACM international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 176–188, Santa Clara, USA, April 1991.

[2] Theo Ungerer, Bortu Robic, and Jurij Silc. Multithreaded processors. *The Computer Journal*, 45:320–348, 2002.

[3] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[4] A. A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. 978-0-12-385251-9. Elsevier, 2005.

[5] MIPS. Programming the MIPS32® 34K Core Family. Technical report, MIPS Technology, 2005.

[6] Infineon TriCore 2. http://www.infineon.com/.

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

[8] C. Bechara, N. Ventroux, and D. Etiemble. Towards a Parameterizable cycle-accurate ISS in ArchC. In *IEEE International Conference on Computer Systems and Applications (AICCSA)*, Hammamet, Tunisia, May 2010.

[9] 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 *IEEE International Workshop on Workload Characterization (WWC)*, pages 3–14, Austin, USA, December 2001.

[10] J. J. Yi and D. J. Lilja. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. 55(3):268–280, March 2006.

[11] J. Cong, K. Gururaj, G. Han, A. Kaplan, M. Naik, and G. Reinman. MC-Sim: An efficient simulation tool for MPSoC designs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 364–371, San Jose, USA, November 2008.

[12] F. R. Boyer, L. Yang, E.M. Aboulhamid, L. Charest, and G. Nicolescu. Multiple SimpleScalar processors, with introspection, under SystemC. In *IEEE International Symposium on Micro-Nano Mechatronics and Human Science (MHS)*, pages 1400–1404, Nagoya, Japan, October 2003.

[13] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb. 2002.

[14] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. Legacy SystemC co-simulation of multi-processor systems-on-chip. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 494–499, Freiburg, Germany, 16–18 Sept. 2002.

[15] S. Cordibella, F. Fummi, G. Perbellini, and D. Quaglia. A HW/SW co-simulation framework for the verification of multi-CPU systems. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 125–131, Lake Tahoe, USA, November 2008.

[16] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *International Euro-Par Conference*, pages 716–726, Toulouse, France, August 1999.

[17] J. J. Sharkey, D. Ponomarev, and K. Ghose. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton, October 2005.

[18] SuperESCalar simulator. http://www.sesc.sourceforge.net/.

[19] D. Nussbaum, A. Fedorova, and C. Small. An overview of the Sam CMT simulator kit. Technical report, Sun Microsystems, Mountain View, CA, USA, 2004.

[20] R. Leupers and P. Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. *Design Automation in Embedded Systems*, 3(1):1–36, November 1998.

[21] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *IEEE European Design and Test Conference (Euro-DAC)*, page 503, Brighton, England, September 1995.

[22] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA—machine description language for cycle-accurate models of programmable DSP architectures. In *ACM/IEEE Design Automation Conference (DAC)*, pages 933–938, New York, USA, June 1999.

[23] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-Based Architecture Description Language. In *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 66–73, Foz do Iguaçu, Brazil, October 2004.

[24] 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 *Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 673–678, Seoul, Korea, January 2008.

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

[26] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. 0123704901. Morgan Kaufmann Publishers Inc., 4th Edition, 2006.

[27] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. SESAM: an MPSoC Simulation Environment for Dynamic Application Processing. In *IEEE International Conference on Embedded Software and Systems*, Bradford, UK, July 2010.

[28] Harvey Mudd College and University of Adelaide. Hmc mips, http://code.google.com/p/hmc-mips. Version 7.3.15, March 2007.