An Inter-core Communication Enabled Multi-core Simulator Based on SimpleScalar

Rongrong Zhong, Yongxin Zhu, Weiwei Chen, Mingliang Lin
Shanghai Jiao Tong University
{zhongrongrong, zhuyongxin, chenweiwei, linmingliang}@ic.sjtu.edu.cn

Weng-Fai Wong
National University of Singapore
wongwf@comp.nus.edu.sg

Abstract

In the recent years, multi-core processors prove their extensive use in the area of System-on-Chip (SoC) on a single chip. This paper proposes a methodology and implements a multi-core simulator. The multi-core simulator is based on SimpleScalar integrated with SystemC framework, which deals with communication and synchronization among different processing modules. A shared memory scheme is introduced for inter-core communication with a set of shared memory access instructions and communication methods. A synchronization mechanism, which only switches the simulation component when communication occurs, is proposed for efficiency. Experiments prove that our simulator can correctly simulate the behavior of a multi-core system and demonstrate a high performance on Linux PC platforms.

1 Introduction

After proven successes on supercomputers, multiple processor systems evolve into Systems-on-Chip (SoC). Multiprocessors have gradually became the backbone of current embedded systems in the areas of network processors [2], multimedia processing [7], security enhancement [9] and mobile terminals [3]. All these areas are characterized by ever growing computation complexities and multi-tasking. The design complexities are further challenged by the time to market and strict cost constrains. To accelerate the design flow and shorten the time to market of multiprocessor systems, many researchers and industrial practitioners have been exploring the fields of system architectures, hardware/software co-simulation, debugging methods and compiler issues for the multiprocessor systems.

Among intensive research efforts on the simulation of multiprocessor systems, most work focuses on the systems consisting of homogeneous processors. The few efforts that address the systems with heterogeneous or hybrid multiprocessors could not properly handle the multitasking issues such as communications among the multiprocessor and multi-task synchronization.

The major problems in simulating heterogeneous or hybrid multi-core processors arise from the difficulties in coordinating different single processor simulators due to the differences in both architecture and implementation. Besides, a well organized communication mechanism is needed for efficient, configurable and accurate simulation.

This paper proposes a simulation methodology with implementation of a multi-core simulator. The multi-core simulator is built with the famous SimpleScalar tool set [4] because it is popular among academic research and supports various existing architectures. We employ SystemC [1] to encapsulate different architecture simulators to provide uniform interfaces. In this way, we can integrate both homogeneous and heterogeneous single processor simulators with only slight modifications as long as they are written in C/C++, and provide possibility for co-simulation with those existing commercial product modules. We implement a shared memory module in SystemC to provide inter-core communication. Communication occurs only when necessary to ensure both accurate synchronization and a better simulating performance. The multi-core simulator also supports bus arbitration and some basic shared memory services.

Section 2 discusses related works of multi-core or multithread simulators. Section 3 introduces the methodology and framework of the proposed multi-core simulator, while section 4 describes the implementation details. Section 5 evaluates the performance with several sets of tests. And Section 6 summarizes this paper.

2 Related Works

Many ongoing efforts on enhancing the simulation of multiprocessor Systems-on-Chip have been reported. Gerin et al. [8] presented a co-simulation framework targeting heterogeneous multiprocessor architectures which represents communication aspects in message level, driver level and register transfer level. Benini et al. [5] proposed a more complete infrastructure with delicate definitions of communication interfaces, which significantly lessen costs of new module integration. The balance of the approaches of Gerin et al. [8] and Benini et al. [5] appears to be more on the communication side as the abstract level of processor cores in their framework is higher than the micro-architecture level. However, the overall performance of a multiprocessor system depends not only on system level communication design, but also on the micro-architecture of each single processor. Therefore, an efficient multiprocessor simulator should have the capability to model both system level and micro-architecture level design.

To exploit design space at both system and micro-architecture level, some researches have been performed by simulating multi-processor based on SimpleScalar tool set. For example, Boyer et al. [6] integrated SystemC wrappers and the SimpleScalar simulator. The strategy that they use is to encapsulate the SimpleScalar simulator in C++ classes and transform it into an open component, communicating with external world. Manjikinn [12] proposed multiprocessor enhancements based on the SimpleScalar simulator. The core simulation code is modified to support multiprocessing, and a run-time library is added for thread creation and synchronization. Jian Huang et al. [10] developed a Simulator for Multithreaded Computer Architectures (SIMCA) based on SimpleScalar. The simulator introduces process-pipelining to hide the details of the underlying simulator, and achieves a performance of only 3.8 to 4.9 times slower than the SimpleScalar based simulator.

The latest extension to simulate multiple cores on a single processor based on SimpleScalar is the work by Donald and Martonosi [11]. They enabled simulation of concurrent execution of multiple cores in a processor by incorporate multiple SimpleScalar engines into one control body. The major difference between their work and ours is that inter-core communications are enabled in our methodology. As such, it is feasible for users of our simulator to simulate cooperations of multiple cores cooperate on a shared task, instead of the independent tasks assumed by Donald and Martonosi [11].

3 Methodology and Framework

3.1 Develop the Framework with SystemC

In this paper, we employ SystemC to build the framework of the multi-core simulator. SystemC has been adopted by EDA community to co-simulate systems described in a mixture of HDL and SystemC. It has been proven that SystemC works well to describe different modules in terms of functionality and hierarchy. We hence take SystemC as the wrapper of multiple cores simulated by SimpleScalar derivatives based on different instruction sets. Additional advantages of SystemC are as follows.

It provides an efficient way to build multi-core simulator. We can develop a multi-core simulator based on any existing simulator that is written in C or C++ languages. The modifications on the original simulator only include providing a uniform single core simulator interface in SystemC and encapsulating the main functions and variables into this module. If the single core simulator is written in C, other functions can be linked into a static lib and remain as external C functions as long as the single core simulator module functions well. This method of building a multi-core simulator can significantly reduce the workload.

A modularized simulator is easier for both development and maintenance. Usually a multi-core simulator may include different cores, hardware accelerators, memory modules, and communication fabric. A SystemC framework can provide a well organized view and accurate timing to make the job much easier.

3.2 Simulator Framework

The simulator framework we proposed is shown in Figure 1. The modules within the frame are already implemented in our simulator. We choose to build the multi-core simulator based on SimpleScalar, the popular instruction set simulator for academic use. Except part of the SimpleScalar simulators that are implemented in C language, all the components are implemented in SystemC which provide interfaces that can be connected by SystemC channels.

In our simulator, we implement a dual-core system based on different instruction-set SimpleScalars, i.e. a PISA instruction-set one and an ARM instruction-set one, a shared memory, and a shared memory bus arbiter dealing with shared memory accesses. Other components such as another core or hardware accelerators can be added to this simulator system for the aim of co-simulation as long as they implement a shared memory bus interface.

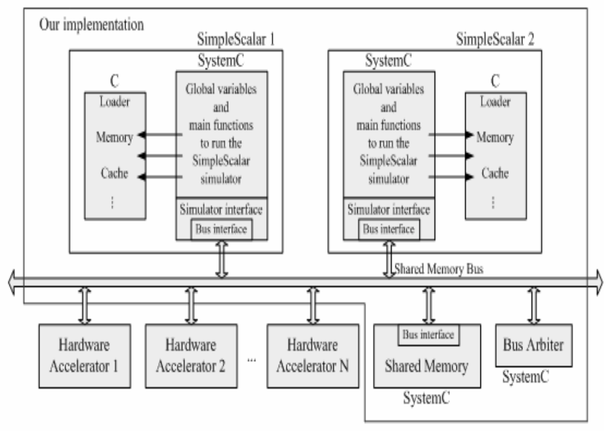

Figure 1. Multi-core Simulator Framework.

3.3 Synchronization Schemes

One of the major difficulties of multi-core simulation with communication is the synchronization among different processing components. Synchronization enables cooperation of multi-core on a single application. If synchronization is not properly handled, it may generate incorrect result.

The safest approach to multi-core synchronization is round-robin, which runs each of the processing components cycle-by-cycle(or instruction-by-instruction, in a functional model) alternatively. However, this approach may cause dramatic performance penalty due to the frequent thread switching between different processing components (which are implemented as SystemC modules). Our experiments show that with two SimpleScalar modules running in cycle-by-cycle round robin approach, the simulation speed drops significantly compared to the original single SimpleScalar processor. Table 1 shows the test results of sim-safe and sim-outorder simulators.We can see that When simulating in sim-safe mode, the performance drops to only 17% to 28% of the original SimpleScalar. Though the result of sim-outorder mode is much better, there is still a penalty of about 33%, even there is no inter-core communication at all.

A better solution is to simulate in a coarse-grained round-robin approach, for example, run every processing component for several cycles and then switch to the next. Although this approach can give a better performance, it may generate incorrect simulation result since one processing component may already run to the 1000th cycle while another processing component tries to communicate with it at the 999th cycle.

A more efficient way is needed for inter-core synchronization to get a better performance while guarantee the accuracy. In this paper, we introduce a communication based synchronization approach which can well satisfy our needs. In this approach, synchronization between different cores is only done when communication is necessary. Since instructions run within one processing component has nothing to do with those run on the other processing components, only instructions that relate to communication between different components, that is, instructions access the shared memory, may cause synchronization problems. Thus we only need to invoke synchronization when shared memory accesses occur. To keep the implementation simple, we choose to integrate this synchronization handler into the bus arbiter module. As illustrated in figure 2, one processing component runs until a shared memory access instruction is decoded. Then the shared memory access request and the simulation cycle count will be sent to the bus arbiter, which compares the current cycle number of all the processing components in the system and grands the one with the smallest cycle number access to the shared memory.

Table 1. Performance Penalty with Round Robin Approach.

Program			jpeg	qsort	sha
speed (k inst. /sec.)	sim-outorder	1-core*	725	401	830
		2-core**	486	261	591
		speed loss	33%	35%	29%
	sim-safe	1-core	9351	5354	10704
		2-core	1014	759	931
		speed loss	78%	72%	83%

* Average of SimpleScalar with PISA and ARM instruction sets.
** Two-core simulator with a PISA and an ARM instruction sets.

3.4 Communication Mechanism

To work with the synchronization scheme, we take shared memory as the inter-processor communication media in our simulator framework. The implementation of the shared memory illustrates the potential to incorporate other communication mechanism. Any processing component that wants to access shared memory should implement special load/store instructions and a functional unit called shared memory port. We assume that all the processing components are connected to the shared memory using a devoted shared memory bus. A devoted bus can ensure fast access and communications, and bus arbitration are provided to avoid conflictions. The shared memory module is written in SystemC.

We also provide a library of shared memory related communication services to facilitate parallel computing tests. The services now only include shared memory $allocation$, $deletion$ and $mail\ box$ services. Other services are to be added in the near future, such as $semaphore$ and

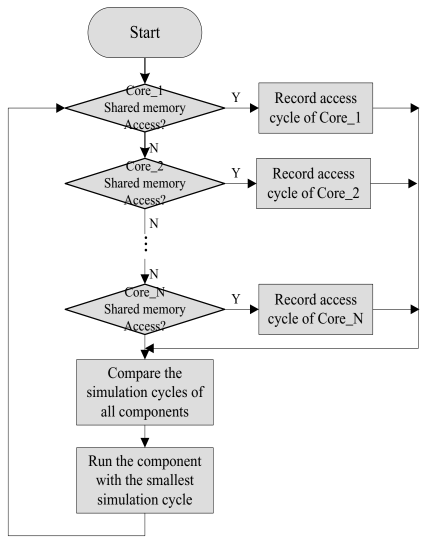
Start
Core_1 Shared memory Access?
Y
Record access cycle of Core_1
N
Core_2 Shared memory Access?
Y
Record access cycle of Core_2
N
N
Core_N Shared memory Access?
Y
Record access cycle of Core_N
Compare the simulation cycles of all components
Run the component with the smallest simulation cycle

Figure 2. Synchronization Scheme Involving the Shared Memory.

message queue.

The services we provide are listed as follows:

1. Shared memory allocation and deletion services:

These services are provided so that an application run on any of the processing components connected to the shared memory bus can allocate a variable into the shared memory space.

. **int ShmAlloc (int type, int size)**

Shared Memory Allocation: Allocate a piece of memory space in the shared memory and return the index *id* of this piece of memory.

. **void ShmDel (int id)**

Shared Memory Deletion: Free a piece of memory space in the shared memory according to the index *id*.

2. Mail box services:

Every processing component connected to the shared memory bus has its own mail box in the shared memory which is used for communication among processors.

. **MBmsg *MBPend (int proc_num)**

Wait for a message from the processing component *proc_num*'s mail box.

. **void MBPost (int proc_num, MBmsg *msg)**

Send a message to the Processing component *proc_num*'s mail box.

3.5 Shared Memory Bus Arbiter

A shared memory bus arbiter is implemented to arbitrate the bus conflictions. Every processing component connected to the shared memory bus has a fixed level of priority. When confliction occurs, the bus arbiter will grant the processing component with the highest priority the accessibility to the shared memory.

4 The Simulator and Implementation Details

4.1 SimpleScalar Architecture Extension

As discussed in section 3.4, if the SimpleScalar cores want to use the shared memory module to communicate with the other components, they must implement the special shared memory access instructions and the shared memory ports as the shared memory bus interface. A shared memory port is integrated into SimpleScalar simply by adding this new functional unit into the SimpleScalar resource. However, the implementation of the instructions is a little more complex.

In order to use these new instructions, first, we need to add these new instruction definitions into the SimpleScalar machine definition. We choose to use dummy instructions, i.e., the instructions that make sense to SimpleScalar compilers but will never be used in applications, as the symbols of our new instructions, thus avoiding the changes to SimpleScalar compiler tools. After the dummy instructions are decoded, we replace the *opcodes*, *flags* and instruction implementations with our own definitions, and send them back to the pipeline. From then on, the instructions will be recognized as shared memory access instructions throughout the pipeline. Memory access latency is calculated according to the feedback from the shared memory bus arbiter.

4.2 Shared Memory Access and Implementation

Usually one shared memory access needs four module-switching, which is shown in figure 3. First, the processing component that wants to access shared memory should send request to wake up the bus arbiter. Then, the arbiter grants one component the right to access and send signal to wake it up. After that, the processing component generates a bus transaction and wakes up the shared memory module. Finally, after the shared memory dealing with the data transfer, the running module is switched back to the processing component. Since the module switching is time-consuming, we integrate the shared memory and bus arbiter into one module in our implementation, which further optimizes the simulator performance. From figure 4 we can see that our implementation only requires two module-switching for each shared memory access. Though such an implementation will not reflect the timing of any physical buses, our simulation model will still generate correct result since SimpleScalar itself does not implement real buses.

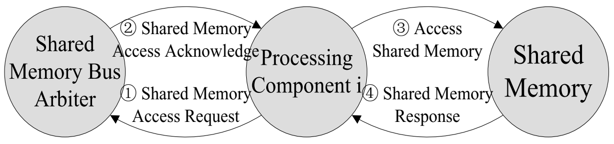
Shared Memory Bus Arbiter
② Shared Memory Access Acknowledge
① Shared Memory Access Request
Processing Component i
③ Access Shared Memory
④ Shared Memory Response
Shared Memory

Figure 3. Four Module-Switching Shared Memory Access.

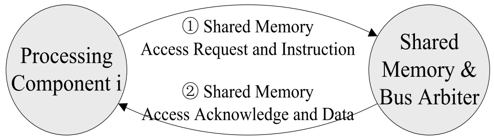
Processing Component i
① Shared Memory Access Request and Instruction
② Shared Memory Access Acknowledge and Data
Shared Memory & Bus Arbiter

Figure 4. Two Module-Switching Shared Memory Access.

Figure 5 shows the implementation of the joint shared memory access module. When a shared memory access instruction is executed, a bus request signal will be sent to wake up this module. Meanwhile, the shared memory access information, i.e., the instruction and address, will also be sent to this module. The shared memory access module will buffer information of one instruction for every processing unit that connects to the shared memory bus. After arbitration, it will grant one processing component the right to access and run its instruction in the buffer. Latency information for every component will be updated at the same time.

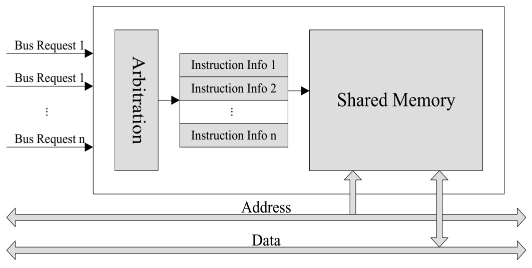

Figure 5. Implementation of the Joint Shared Memory Access Module.

5 Experiments and Evaluations

In order to evaluate the performance of the multi-core simulator, we conduct three test sets. The first test set aims at evaluating the simulator performance without communication overheads. We run the same benchmarks on both cores without any communication whose results are shown in Table 2.

Table 2. Multi-core Simulator Run the Same Program without Communication.

Program		jpeg	qsort	sha
Simulation time (seconds)	SingleSS1*	116	86	146
	SingleSS2**	168	115	189
	SS1+SS2	284	201	335
	Multi-core***	283	201	334
Simulation speed (k inst./sec.)	SingleSS1*	872	568	941
	SingleSS2**	623	349	745
	SS1+SS2	725	401	830
	Multi-core***	727	480	833

* Simplecalar with PISA instruction set.
** SimpleScalar with ARM instruction set.
*** Two-core simulator with a PISA and an ARM instruction sets.

We would like to stress that the multiple processes for the multiple cores in our simulator actually share the CPU time of the single core processor on our PC. In other words, our simulator actually executes the benchmark for multiple times on the single core processor on our PC. Therefore, we should compare our results against the sum of execution time of stand alone runs. Even if there are multi-core processors PC available, there is no effective compiler to assign a simulator child process to a specific core.

In Table 3, the simulation time of our multi-core simulator is no less than the sum of individual runs without communication or synchronization. This indicates the efficiency of our synchronization scheme. The efficiency is further illustrated by the higher simulation speed of our multi-core simulator.

Table 3. Tests for Communication Overheads.

data memory access (KB)		500	1,000	5,000
simulation speed (k inst./sec.)	SingleSS1*	688	687	699
	SingleSS2**	502	603	581
	Multi-core***	458	465	463
speed lost(%)		9.5-33.3	22.9-32.3	20.3-33.7

* Simplecalar with PISA instruction set.
** SimpleScalar with ARM instruction set.
*** Two-core simulator with a PISA and an ARM instruction sets.

Since the synchronization scheme is application sensitive, we are also concerned with the worst case of performance. The second test set consists of programs that frequently read and write shared memory, i.e., programs first write certain amount of data to the shared memory and then read from it. The test results are compared with results of the multi-core simulator when each of its processing components runs a program that read and write its own memory with the same problem size. From the result shown in Table 3 we can see that we may have a performance penalty of about 33% due to frequent communication. Since usually there will not be this much communication between different cores, we can expect a better performance for real applications.

The last test evaluates the simulator performance under a common embedded application, i.e., an mp3 decoder. We adopted an mp3 decoder to the multi-core environment by dividing the decoding into two stages (stage one including Sideinfo Extracting, Huffman Code Decoding, Requantization, Reordering, Alias Reduction and IMDCT; the seconde stage including Polyphase Synthesis Filter which asks for about 50% calculation complexity in the whole mp3 decoding algorithm), each of which runs on one core. Though this application is not well partitioned, and there are minor errors in the MP3 decoding process due to the compiler incompatibility (the MP3 decoding program we adopted was originally designed for ARM), as long as these errors do not affect the timing metrics, it still serves as a good case of evaluating the simulator performance. Table 4 shows the test results of decoding 2 frames, 5 frames and 10 frames of a MP3 file. The application speed up due to parallelism is round 22%, while the speed lost due to communication and synchronization is less than 9%. Since a poor partition will cause more overheads on synchronization, we can expect a better performance when using a better partition.

Table 4. MP3 Decoder Simulation Results.

Number of Frames		2	5	10
simulated cycle	Single core	8.297e+7	2.140e+8	4.336e+8
	Multi-core	7.432e+7	1.581e+8	3.381e+8
	Speedup	10.43%	21.47%	22.03%
simulation time (sec.)	Single core	232	599	1207
	Multi-core	276	685	1358
	Overhead	18.97%	14.36%	12.51%
simulation speed (k inst./sec.)	Single core	526.435	529.320	530.646
	Multi-core	494.167	486.821	484.065
	Speed loss	6.13%	8.03%	8.78%

6 Conclusions

In this paper, we have implemented a multi-core simulator based on SimpleScalar with framework, communication and synchronization implemented in SystemC. The simulator adopts shared memory, with shared memory access instructions and communication methods, as the inter-core communication scheme. In order to speed up the performance, we introduce a synchronization mechanism which only requires module switching when communication is necessary. This simulator can correctly simulate multi-core system and provide extensibility for co-simulation with other modules written in SystemC.

Future work may include extensions of shared memory services, cacheable shared memory module implementation, other cores and hardware accelerators integration which will enable a more comprehensive and flexible co-simulation.

References

[1] Open systemc initiative (OSCI), functional specification for systemc 2.0, 2001. http://www.aystemc.org.

[2] Product brief. Intel IXP2850 network processor, 2002. http://www.intel.com.

[3] A. Artieri, V. Dalto, R. Chesson, M. Hopkins, and C. Marco Rossi. NomadikTM open multimedia platform for next generation mobile devices, 2003. http://www.st.com.

[4] T. Austin, D. Burger, G. Sohi, M. Franklin, S. Breach, and K. Skadron. The simplescalar architectural research tool set, 1998. http://www.cs.wisc.edu/mscalar/simplescalar.html.

[5] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. Systemc cosimulation and emulation of multiprocessor SoC designs. *Computer*, 36(4):53–59, April 2003.

[6] F. Boyer, L. Yang, E. Aboulhamid, L. Charest, and G. Nicolescu. Multiple simplescalar processors, with introspection, under systemc. *Circuits and Systems, Proceedings of the 46th IEEE International Midwest Symposium*, 3:1400–1404, December 2003.

[7] S. Dutta, R. Jensen, and A. Riechann. Viper: A multiprocessor SoC for advanced set-top box and digital TV systems. *IEEE Design and Test of Computers*, 18(5):21–31, September-October 2001.

[8] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya. Scalable and flexible co-simulation of soc designs with heterogeneous multi-processor target architectures. *Proc. Conf. Asia and South Pacific Design Automation (ASP-DAC 01), ACM Press*, pages 63–68, 30 Jan.-2 Feb 2001.

[9] J. Helmig. Developing core software technologies for TI's OMAPTM platform, 2002. http://www.ti.com.

[10] J. Huang and D. Lilja. An efficient strategy for developing a simulator for a novel concurrent multithreaded processor architecture. *Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 185–191, July 1998.

[11] D. James and M. Margaret. An efficient, practical parallelization methodology for multicore architecture simulation. *IEEE Computer Architecture Letters*, 5(2):14–14, Feburary 2006.

[12] N. Manjikian. Multiprocessor enhancements of the simplescalar tool set. *ACM SIGARCII Computer Architecture News*, 29(1):8–15, March 2001.