A Soft Methodology for On-Chip Multiprocessor Design

Alejandro Matute[1], Marcos de Alba[1], and Elias Mizan[2]

[1] Tecnológico de Monterrey, Departamento de Ciencias Computacionales
Atizapán, Estado de México, México 52926
(A00464063, marcos.de.alba)@itesm.mx
[2] The University of Texas at Austin, Electrical and Computer Engineering Department
Austin, TX 78751
emizan@ece.utexas.edu

Abstract. We propose a software methodology towards the design of on-chip multiprocessors at the architecture level. This methodology will be helpful to explore multiprocessor characteristics like reliability, security, redundancy and soft-error detection and correction, among others. The technique proposed provides a clean design without loss of functionality or performance. One of its advantages is that it is scalable for the design of on-chip multiprocessors with up to n cores. We explain its application with two architecture organizations. In the first, two or more CPU cores are synchronized by executing their own applications in bulks of the same number of instructions at each core. In the second, CPU cores execute applications performing as many instructions as possible without synchronizing.

1 Introduction

The design and implementation of fully-reliable synchronization strategies between microprocessor cores is an important requirement in multiprocessor design. Architect-level simulation is one of the initial steps for multiprocessor design it provides flexibility for fast update of processor components in software, speeding up the evaluation of ideas allowing architects to explore the design space. The lack of correct multiprocessor simulators can cause misleading communication between CPU cores, affecting performance, functionality that can lead to wrong conclusions.

In the design of a superscalar microprocessor a simulator like [1] is very helpful for exploring its design space. To design the architecture of an on-chip multiprocessor we utilize a uniprocessor simulator and make modifications to it. There are different ways to accomplish this; we believe that the methodology we propose is the more natural and efficient regarding programming and functionality. This suggests replicating simulated superscalar microprocessors through multiple operating system level processes. This means that the entire operating system context of a superscalar microprocessor simulation is replicated, so that any multiprocessor architecture can be designed by multiple replicated uniprocessors.

To provide communication among different uniprocessors, or CPU cores for simplicity, we define shared variables. For example, a four-core symmetric multiprocessor, SMP, can be designed generating four identical OS-level processes of the uniprocessor simulator and defining a set of shared variables that represent shared hardware resources among the CPU cores, like a level-two unified cache. To explore replacement and writing policies on shared resources it is proposed the definition of programming subroutines which parameters include shared resources. These are implemented straightforward using OS mutual exclusion techniques.

In this work we define two multiprocessor architecture schemes. In the first, N instructions are executed at each CPU core, to assure this a synchronization mechanism is defined. In the second, every core executes freely, that is, without having to synchronize.

2 Related Work

There has been a great amount of multiprocessor design research during the last decades. Multiprocessors were originally built by interconnecting several computers or by interconnecting several CPU cores. More recently, due to the large increase on transistor density researchers have integrated two or more CPU cores on a single chip to provide higher performance, more reliability and better power efficiency.

To explore multiprocessor design, there have been proposed different strategies, including: Theoretical approaches, interconnected-CPU multiprocessors, chip multiprocessors, simulators and emulators.

Since the focus of this work is the description of a software methodology for multiprocessor design, we do not discuss hardware-based multiprocessor approaches.

There are available multiprocessor simulators like [3-6]. However, they are targeted for designing off-chip multiprocessors. In [11] it is mentioned an architecture level multiprocessor simulator, but it is not available and no further information can be obtained. Thus, we decided to concentrate on the architectural design of on-chip multiprocessors developing our own simulation infrastructure using [11] as a baseline.

As described in [2], we are exploring techniques to improve the design of on-chip multiprocessors and we have utilized the methodology explained in this paper as the main baseline simulation technique to explore multiprocessor characteristics like reliability, security, redundancy and soft-error detection and correction, among others.

The rest of the paper is organized as follows. In Section 2 we briefly explain the two previously defined simulation schemes for a multiprocessor system and the disadvantages of its loop-based solution. In Section 3 we describe their software implementation and in Section 4 we close with some results that validate our methodology.

3 Multiprocessor Schemes

The multiprocessor schemes we propose are described next.

3.1 Free running

n microprocessor cores run in parallel different applications. They execute different kinds of instructions and share input/output modules. Synchronization is required 3for a shared memory organization. A simulation of this scheme provides fast response, which is suitable for performance or throughput studies on a multiprocessor design. An example of a dual-core multiprocessor with this scheme is shown on Figure 1.

3.2 N instructions per core

n microprocessor cores run the same application in parallel. In this scheme, every core executes the same instructions in chunks of N. That is, at the beginning of an executed simulation in a dual-core multiprocessor, core 0 executes the first N instructions, and then core 1 executes the first N instructions. After this, core 0 executes the next N instructions of the application and core 1 the same next N instructions. Later on, core 0 will execute the third set of N instructions of the program and so on. The purpose of this scheme is to replicate the execution on a given number of cores; this organization is very helpful for investigating reliable microprocessors.

The intention of replicating code execution is to catch up when an execution error occurs on the leading core and being capable of detecting it on the trailing core to correct the execution of the leading core. Figure 2 represents an organization with this scheme.

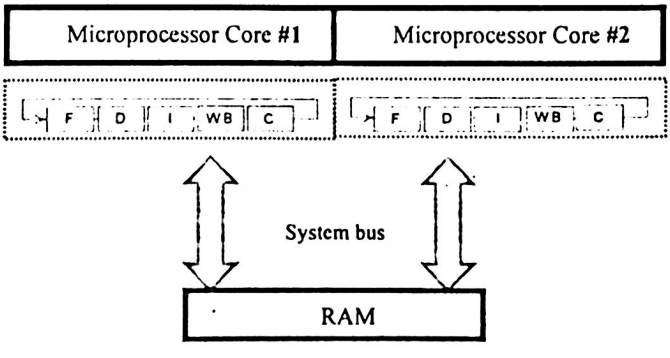

Fig. 1. Free running multiprocessor scheme. N microprocessor cores execute simultaneously without synchronizing

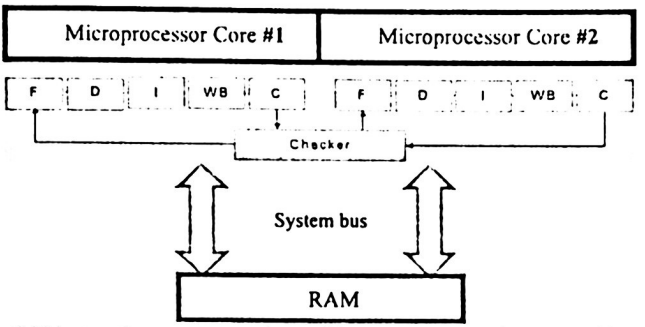

Fig. 2 N instructions per core scheme. The same program is executed by two microprocessor cores each executing the same chunks of N instructions

3.3 Loop-based scheme

Another scheme to provide a multiprocessor simulation environment is to replicate the uniprocessor resources by declaring arrays instead of single variables. However, comparing this approach with the previous two we noticed the same result would be produced without modifying all source code files. Therefore, maintainability and functionability would be improved. We also decide not to use this approach to keep instruction-level execution synchronization control on cores.

4 Software Implementation of Multiprocessor Schemes

In this section we discuss how we implemented the multiprocessor schemes on a base superscalar microprocessor simulator to produce a reliable simulator for on-chip multiprocessors. Our goal is to demonstrate that this is an easy-to-adapt architecture-independent mechanism for any uniprocessor simulator which provides a more scalable and flexible architecture level analysis of uniprocessor details (like bus traffic, cache sizes or registers characteristics), unlike other multiprocessor simulators that focus their analysis details on the inter-processor communication. We also believe this approach could be considered as a good option when other multiprocessor simulators like SimpleScalar Multiprocessor Simulator are found to be not available.

The operating system we use is Linux. The base simulator has a binary file that is assigned to an operating system process. By replicating the running process of the simulator [1] at the beginning of the simulation, two or more microprocessor cores would be easily emulated with very few changes in the original code. The proposed methodology consists on generating multiple processes of the same executable through *fork()* calls to the operating system.

The simplicity of this implementation keeps the functionality and performance of the simulator and provides a scalable solution. By just making *n* calls to *fork()* n CPU cores will be running the same application.

4.1 Free running implementation

The free running implementation requires no synchronization, since every core executes instructions freely. A shared memory implementation of this scheme would require declaring shared memory variables and the addition of cache coherence protocols like [7] or [8]. Our initial approach was to initially explore a non-shared memory multiprocessor organization.

4.2 N instructions per core implementation

A more common multiprocessor organization has shared memory among the available CPU cores. To implement this scheme on the simulator we designed the semaphores described in [9] in order to synchronize the cores and declared shared variables that represent shared hardware resources. With additional shared memory this type of organization can be used to model on-chip multiprocessors with soft-error detection and recovery techniques.

The semaphores provide a source to block a microprocessor core until the next one completes a given set of instructions.

Assuming a multiprocessor with two cores, the use of a single semaphore can control their execution, the software implementation of this synchronization scheme is represented in Figure 3.

In Figure 3 the block named Code #1 represents the instructions to be executed by the core 1. The block Code #2 represents the instructions to be executed by the core 2. The execution of Code #2 depends on whether core 1 has completed executing Code #1; if it has not, then core 2 will be stalled until core 1 completes its work. Similarly, Code #1 depends on the execution of Code #2 on the next round.

To synchronize multiprocessor cores at the instruction level, we set the control barrier before the commit stage of the pipeline in each core. The code fragment shown on Figure 4 demonstrates this idea on software.

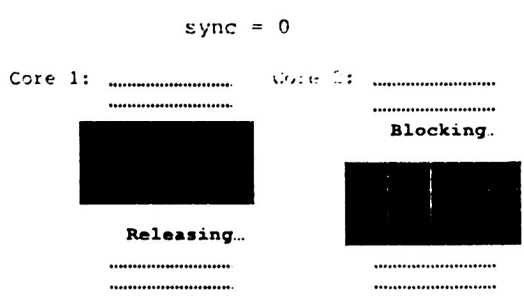

Fig. 3 Semaphore-based synchronization mechanism

```
//instruction pipeline
begin
if (pid == MICRO_0) P(SEM0);
else P(SEM1);
instr_commit();
if (pid == MICRO_1) V(SEM0);
else V(SEM1);
...
instr_issue();
...
end
```

Fig. 4 Software implementation of a synchronized pipeline

Notice that the previous code could also be implemented by replacing the conditional sentences by a hash table to store semaphore IDs and using every CPU core ID as the key to access the table, providing a scalable scheme up to *n* microprocessor cores.

A hardware implementation of a synchronized pipeline would look like that shown in Figure 5.

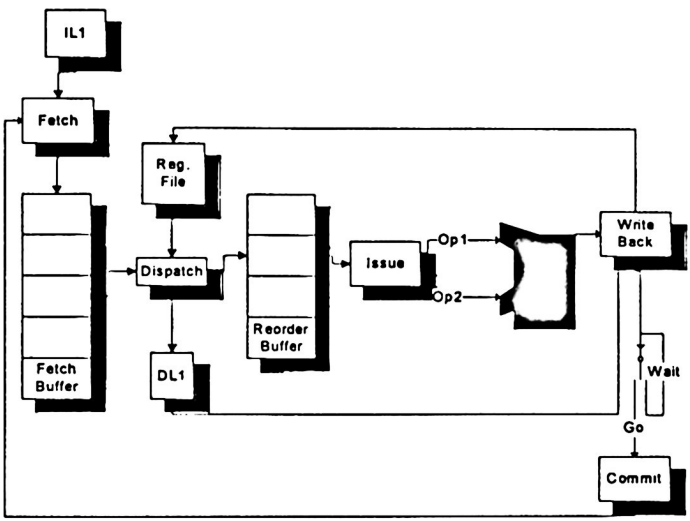

Fig. 5 Hardware implementation of a synchronized pipeline

On Figure 5, the semaphore acts like a double pole switch. On the GO position the pipeline allows the flow of data from commit to fetch. On the STOP/WAIT position the pipeline is stalled until the controlling core executes its corresponding set of instructions. This synchronizing mechanism allows multiple cores to work coordinated.

5 Experimentation and Results

To validate our methodology we executed simulations with a subset of SPECint2000 on our multiprocessor simulator derived from the superscalar microprocessor simulator [1]. For all the benchmarks listed in Table 1 we simulated 100 million instructions.

Table 1. Benchmarks and dataset inputs used for simulations

Benchmark	Input	Benchmark	Input
Bzip2	Input.graphic	**vpr**	net.in, arch.in, place.in
Gcc	integrate.i	**crafty**	crafty.in
Gzip	Input.graphic	**gap**	ref.in
parser	ref.in	**mcf**	inp.in
twolf	ref	**perlbmk**	perfect.pl

We first simulated a superscalar uniprocessor, which characteristics are listed on Table 2. These characteristics are similar to those found in a modern microprocessor like [10]. Then, we simulated a dual-core multiprocessor to verify its correctness.

Table 2. Configuration characteristics of superscalar microprocessor

Component	Config.	Component	Config.
Fetch width	4	L1 D-cache	16KB, 4-way
Branch prediction miss penalti	3 cycles	L2 D-cache	256KB, 4-way
Branch predictor access time	1 cycle	I-Cache	16KB, direct
Branch predictor type	Bimodal	Mem. Latencies	18, 2 (cycles)
Branch prediction table size	2048	Bus width	8
Two-level branch prediction table conf.	1 1024 8 0	I-TLB	4KB
Call/return stack size	8	D-TLB	4KB
BTB size and associativity	512 4	TLB miss penalty	30 cycles
Dispatch width	4	Int ALUs	4
Issue witdth	4	Int Multipliers	1
Write back width	4	Memory ports	2
Instruction window size	16	FP ALUs	4
Load/store window size	8	FP Multipliers	1

The results of the simulations are shown in Figures 6 to 9. Figure 6 shows the performance rate of the uniprocessor as the number of committed instructions per clock cycle (IPC). Figure 7 shows the miss rate of the first level data cache on the baseline superscalar uniprocessor. The miss rate is defined as the ratio of number of misses among all the requests to the first level data cache. Figure 8 shows the performance rate of each processor core on the on-chip multiprocessor. Figure 9 shows the miss rate of each of the first level data caches in the multiprocessor. As can be seen the results on Figure 6 and 8 are identical as those in Figures 7 and 9. As expected, our results were equal, because the characteristics in the core processors are equal to those of the uniprocessor and each core in the multiprocessor executes the same benchmark. We are assuming ideal inter-core communication conditions. A communication model with no ideal conditions would impact in the multiprocessor's performance. However, the miss rate of the first level data cache would not be affected, because we have defined a first level data cache per core. We are aware of the fact that considering a shared second level data cache would impact the first level data cache miss rate, so we will explore this in future work. As stated above, the purpose of this multiprocessor configuration is to provide execution redundancy in order to achieve multiprocessor characteristics like reliability, security, redundancy and soft-error detection and correction using this methodology as the main baseline simulation tool used in [2].

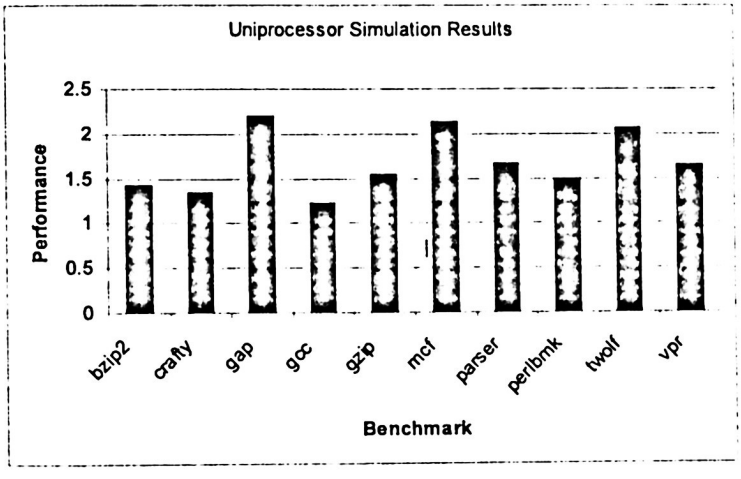

Fig. 6 Performance results of superscalar microprocessor (number of committed instructions per clock cycle)

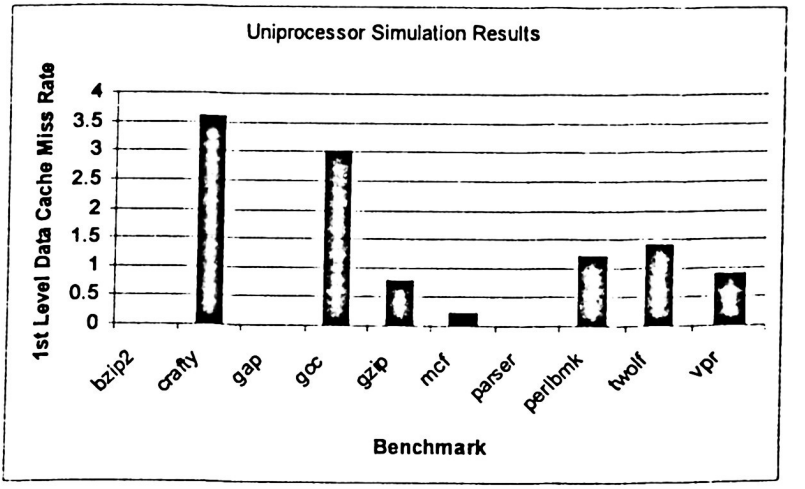

Fig. 7. First level data cache miss rate (%) results of superscalar microprocessor (ratio of misses among all the requests to the first level data cache)

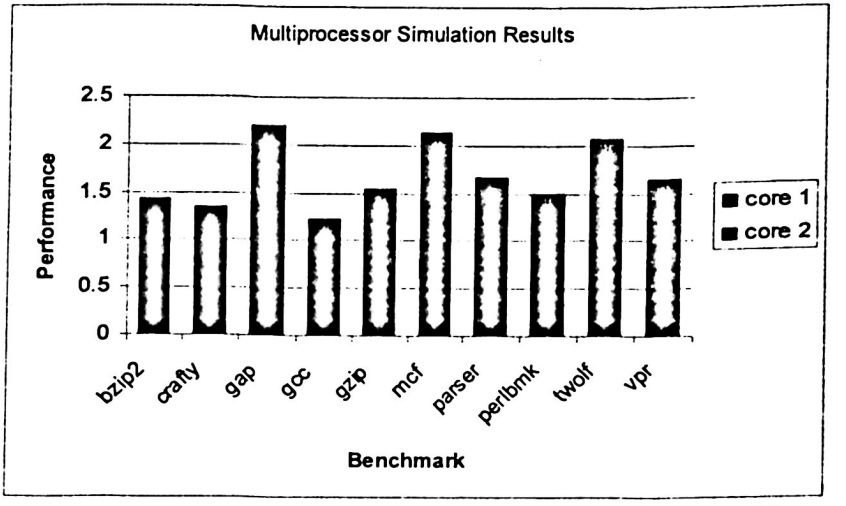

Fig. 8 Performance results of cores in dual-microprocessor (number of committed instructions per clock cycle)

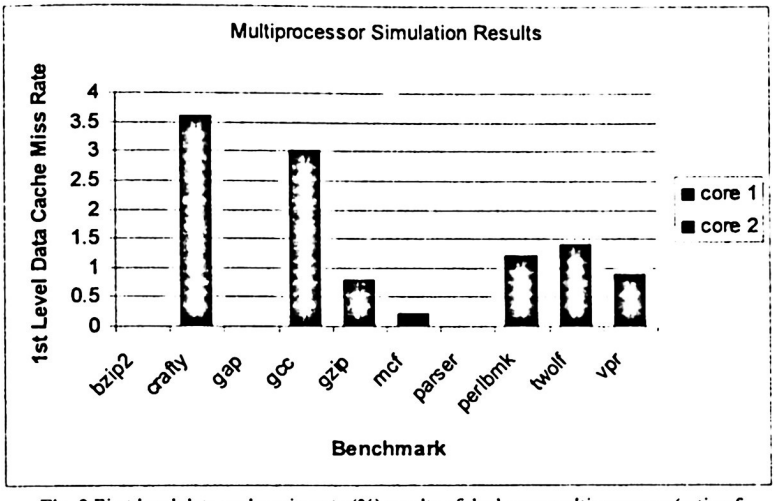

Fig. 9 First level data cache miss rate (%) results of dual-core multiprocessor (ratio of misses among all the requests to the first level data cache)

6 Conclusions and Future Work

We have proposed a methodology for on-chip multiprocessor design using a uniprocessor simulator as a baseline and operating system process replication functions along with shared memory variables. Our methodology is simple and useful to explore new architectural ideas for on-chip multiprocessors. We demonstrated that the multiprocessor simulator provides the same results as the uniprocessor simulator, but on multiple processor cores under ideal inter-core communication conditions, which is a very useful technique for investigating soft-error detection and correction as well as for exploring fault-tolerant multiprocessors.

The next step in our research is to investigate shared resources in hardware and to evaluate new sharing policies among them.

As shown in [2], we have also proved a technique for the design of fault-toleratn on-chip multiprocessors and we have found this methodology very useful. It has allowed us to explore multiprocessor characteristics like reliability, security, redundancy and soft-error detection and correction.

7 References

[1] Larson, E., Chatterjee, S., Austin, T. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. On ISPASS 2001. Tucson, Arizona.

[2] Mizan, E. and de Alba, M. "Fault-Tolerant CMP Design Using a Write Cache Checker", on DSN 2005 The International Conference on Dependable Systems and Networks". Yokohama, Japan. July 2005.

[3] Pai, V. S., Ranganathan, P. and Adve, S. RSIM: Rice Simulator for ILP Multiprocessors. http://softlib.rice.edu/rsim.html

[4] Matloff, N., Rich, K. Mulsim: Multiprocessor Simulator. http://heather.cs.ucdavis.edu/~matloff/MulSim/MulSimDoc.html

[5] Sunada, D., Glasco, D. and Flynn M. ABSS: SPARC multiprocessor simulator. In proceedings of the 8th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI '98).

[6] Chen, X. Hong, J. Gao, Y. Li, X. Zheng, S. Design and Implementation of Multiprocessor Simulator Simdsm, Minimicro Systems, Shenyang, 2000. Vol. 21; Part 2, pp. 186-189.

[7] Archibald, J. and Baer, J.L. Cache Coherence Protocols: Evaluation using a Multiprocessor Model, ACM, Trans. On Computer Systems 4:4, 1986, (November), pp. 273-298.

[8] Agarwal, A., Simoni, R., Hennesy, J., Horowitz, M. An Evaluation of Directory Schemes for Cache Coherence, In Proc. Of the 15th ISCA, June 1998, pp. 280-289.

[9] Gómez, C., R. Memoria compartida, semáforos y colas de mensajes. Tecnológico de Monterrey, 2002, pp. 16.

[10] Hinton, Glenn (et. al.). The Microarchitecture of Pentium® 4 Processor.

[11] The Simplescalar Web Site. http://www.simplescalar.com.