CATS: Cycle Accurate Transaction-driven Simulation with Multiple Processor Simulators

Dohyung Kim[†] Soonhoi Ha[§] Rajesh Gupta[†]

[†]Department of Computer Science and Engineering
University of California, San Diego, USA
{dhkim, rgupta}@ucsd.edu

[§]School of Computer Science and Engineering
Seoul Nation University, Korea
sha@iris.snu.ac.kr

Abstract

This paper focuses on enhancing performance of cycle accurate simulation with multiple processor simulators. Simulation performance is determined by how often simulators exchange events with one another and how accurately simulators model their behavior. Previous techniques have limited their applicability or sacrificed accuracy for performance. In this paper, we notice that inaccuracy comes from events which arrive between event exchange boundaries. To solve the problem, we propose cycle accurate transaction-driven simulation which maintains event exchange boundaries at bus transactions but compensates for accuracy. The proposed technique is implemented in a publicly available CATS framework and our experiment with 64 processors achieves 1.2M processor cycles/s (200K instructions/s) which is faster than other cycle accurate frameworks by an order of magnitude.

1 Introduction

Multiple processors with simple architecture possibly provide demanding computation power in energy efficient way. In addition, deep submicron technology makes it easier to integrate more processors into a single chip. However, software development for multiple processors becomes more difficult than the case for a single processor. The design space for multiple processors and an inter-connection network becomes incredibly large. To handle those situations, we need methods to evaluate architectural design accurately and efficiently.

Moreover, simulation in embedded systems also has to verify worst case scenarios where a system may not meet its constraints such as deadlines and response times. Methods that sacrifice timing accuracy for speed can not guarantee meeting constraints. Thus this paper focuses on techniques to enhance simulation performance with multiple processor simulators while maintaining accuracy.

To identify performance problems, we analyzed major factors of simulation time in multiprocessor simulation. First, a multiprocessor simulator usually has a single code to model processor behavior and switches different processor instances. Thus, when we invoke a processor, the instance switch generates additional overheads compared to a single processor simulator. Second, we have to execute behavior of processor simulators and an inter-connection network simulator by advancing their clocks. However, a simple processor simulator can adopt a static delay model for an inter-connection network.

When we optimize those two factors of multiprocessor simulation, granularity and abstraction of simulators have major effects to simulation performance. Granularity indicates how often simulators exchange event with one another. Abstraction determines how accurately simulators model behavior of processors or an inter-connection network. If we can have larger granularity, the number of instances switches is reduced. If we can apply higher abstraction, it reduces time to advance clocks of simulators.

Previous approaches have utilized granularity and abstraction to overcome performance problems of multiprocessor simulation. Software analysis predicts [7] arrival times of events by statically analyzing instructions of other simulators. If we know when the next event arrives at a simulator, we can safely advance the clock of the simulator until that time. This technique increases granularity of simulators, which reduces the number of instance switches. Virtual synchronization [8] suspends to process events to a simulator by analyzing behavior of the simulator as long as events do not affect behavior of the simulator. If a simulator waits for certain events, the simulator can skip clock cycles until the simulator receives any of those events. Thus the technique reduces simulation time related to instance switches and behavior model. Transaction level model (TLM) [9][10] abstracts behavior of a simulator by simplifying transitions through multiple states between abstraction boundaries. TLM reduces all components of simulation time but may sacrifice accuracy when a simulator receives any event which affects behavior of the simulator between abstraction boundaries.

All three approaches try to enlarge granularity between boundaries. However, the first and the second approaches have limited their applicability because a static analysis does not predict dynamic behaviors of caches or operating systems. The third approach may sacrifice accuracy for performance.

We notice that three approaches are complementary and propose a novel *cycle accurate transaction-driven simula-*

tion (CATS) technique. In the proposed technique, like the first and second approaches, we predict incoming events to a shared bus and advance clocks of simulators based on the prediction, which prevents accuracy sacrifices of the third approach. Moreover, an extension to the third approach is applied to provide accurate timings of outgoing events to processors. The extension preserves cycle accuracy with little overhead. Therefore, CATS enhances simulation performance by increasing granularity and abstraction. However, it maintains cycle accuracy and does not limit applicability unlike previous approaches.

In addition, *OS sensitive scheduling technique* further reduces simulation time of processors by skipping idle cycles due to interactions between application tasks similar to the second approach.

The proposed techniques are implemented in a CATS framework [1]. The CATS framework extends SimpleScalar [2] to support multiple processor simulators. It also supports a scriptable architecture description and program interfaces for a multithreaded programming model. In the absences of benchmarks for embedded systems supporting multiple processors, we port Splash-2 benchmark [3] to evaluate the CATS framework. Our experiments show that the CATS framework achieves 200K instructions/s with 64 processors in a RADIX application of Splash-2 benchmark.

The main contribution of this paper is to provide a very fast but accurate multi-processor simulation framework which is available as an open source to other researchers. In this paper, we limit our focus on an optimization technique for a shared bus connected to processors with dedicated caches. We are extending the framework to support architectures including multiple buses and network on a chip.

In section 2, we explain related work to enhance simulation performance. Section 3 and section 4 show cycle accurate transaction-driven simulation and OS sensitive scheduling respectively. We introduce the CATS framework in section 5 and demonstrate utility of the proposed technique in section 6. Finally we conclude this paper with future work.

2 Related Work

Many cycle accurate simulation frameworks for multiple processor simulators adopt event-driven simulation [4][5]. Event-driven simulation shows good performance when a simulation has sparse events. However, in the framework for multiple processors, simulators are triggered by every clock cycle. Therefore, overheads to schedule events and executions of simulators are relatively large.

When simulators have static periods, cycle-driven simulation [6][7] builds a static scheduler to minimize the overhead from event-driven simulation. Moreover, using a function call to change instances between processors, overhead related to instance switches also is reduced to a minimal level.

However, approaches which maintain cycle accuracy have had limitations to enhance simulation performance around 100K~300K processor cycles/s because the maximum granularity is a granule of a clock cycle.

Software analysis [8] predicts event arrival times to a shared bus by analyzing application behavior with compiler techniques. Based on the predictions, the technique advances clocks of processor simulators until the next smallest prediction time. The technique reduces instance switches for processors but can not handle dynamic behavior from cache and an operating system.

Virtual synchronization [9] enlarges granularity of simulators to data exchange boundary of application tasks. The technique utilizes application behavior by overriding function calls which transfer data between tasks. Assuming that delays from resource conflicts always stall pipeline stages of a processor, the technique advances the clock of a processor until a task exchanges data with other tasks while capturing resource access traces. At each boundary of event exchanges, trace-driven simulation using resource access traces calculates delays from resource conflicts. Then the delays are inserted into processors as pipeline stalls. Virtual synchronization increases granularity of simulators to data transfer functions and reduces simulation time of processors by skipping cycles when a processor waits for a data transfer. However, due to its assumption, it does not handle events from cache coherence protocol and complex processor models with multiple issued out-of-order instructions.

Transaction level model (TLM) [10][11] abstracts behavior of an inter-connection network at bus transaction level and exchanges events at bus transaction boundaries. With abstracted behavior of simulators, it assumes that simulators do not have events which affect behavior of the simulator between bus transaction boundaries. TLM achieves good performance but suspends to process events between bus transaction boundaries. Thus TLM with higher abstraction may suffer from events from interrupts and cache coherence protocol.

Approaches which enlarge granularity more than a clock cycle shows competitive performance but do not handle some of following events which affect behavior of processors or an inter-connection network; interrupts, unlocked bus transaction, early resuming of pipeline stages and cache coherence protocol. In this paper, we are to solve how to enlarge granularity and abstract behavior by handling those events.

3 Cycle Accurate Transaction-driven Simulation

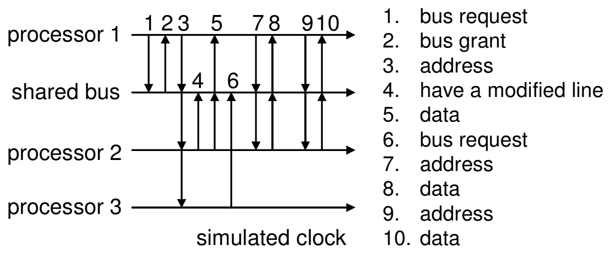

Figure 1. An example of a bus transaction

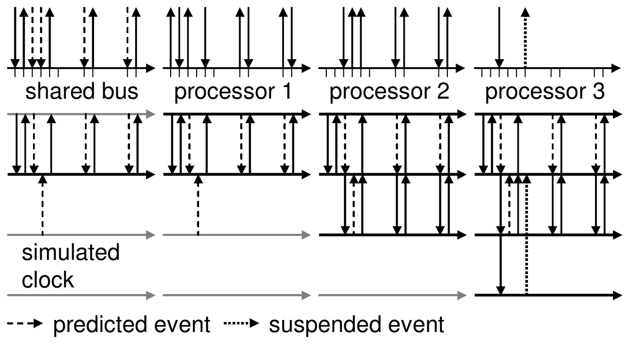

Figure 2. An execution sequence of simulators applying CATS

CATS enlarges granularity of simulators to bus transaction and abstract behavior of a shared bus at bus transaction level similar to TLM. However, events which arrive between granularity boundaries may cause inaccuracy.

Figure 1 shows an example of such a bus transaction. In the transaction, the processor 1 reads data in the modified cache line of the processor 2 and the processor 3 requests a shared bus. When the processor 1 sends the first address of the cache line, the processor 2 interrupts behavior of the shared bus and provides data to the processor 1 instead of a memory. Moreover, the shared bus has to handle the bus request event from the processor 3.

Figure 2 shows an execution sequence of simulators applying CATS, which processes the bus transaction in Figure 1. Upper figures show what events each simulator has during the bus transaction. Bottom figures illustrate how each execution of a simulator incrementally appends events to the bus transaction.

Before CATS advances clocks of simulators, it examines behavior of processors to predict events to the shared bus during the bus transaction. Unlike previous approaches, CATS not only predict arrival times of events but also values of them. Thus the shared bus simulator can advance its clock without interactions with other processors. In addition, behavior of the shared bus enables CATS to suspend a certain type of events to the shared bus. In the example, CATS can predict event 3, 4, 7 and 9, and suspend the event 6.

With predicted events, first, CATS advances the clock of the shared bus until the shared bus finishes processing the bus transaction. During the execution, CATS captures timings of events. Then when CATS advances the clocks of processors, the timings enable processors to emulate incoming events from the shared bus.

Compared to Figure 1, Figure 2 shows an identical simulated result except that CATS does not notify the arrival of the event 6 to the shared bus. However, because we know that the shared bus can suspend the event 6, the difference does not hurt cycle accuracy.

In following sections, we explain the proposed technique in detail. Section 3.1 shows how CATS predicts and suspends events. Section 3.2 introduces how to capture timings of events and handle those timings at processors.

3.1 Handling Events to a Shared Bus

A shared bus receives bus request events, address events (or address events with data) and cache coherence events from processors.

First, a bus arbiter in a shared bus accepts bus request events and grants the bus to one of requesting processors. If a bus transaction locks the bus, the bus arbiter performs arbitration at the end of the bus transaction. Otherwise, the bus arbiter performs arbitration at the end of every memory transaction of the bus transaction.

CATS utilizes such behavior of the bus arbiter and watches a locked flag from a bus transaction. If the locked flag is set, CATS suspends bus request events until the shared bus finishes processing the bus transaction. Otherwise, it suspends requests until a memory transaction ends.

Second, when a processor receives a bus grant, it sends the first address of a bus transaction at the next bus cycle. If the processor receives a data and burst accesses of the bus transaction are not finished, it sends the next address at the next bus cycle. With this property of processors, CATS predicts arrival times and values of address events.

Third, when a processor watches an address which the processor has in its cache line, it sends a cache coherence event to a shared bus and then the shared bus changes its behavior to handle the event. For example, when the shared bus meets a read access to a modified cache line (or a shared cache line), the shared bus makes a requesting processor read data from a processor with that cache line instead of a memory. For a write access to a modified cache line, the shared bus makes a requesting processor first read data from the processor with the cache line and then send a write invalidate event (or update data) to all processors with the cache line.

CATS examines all caches of processors for the cache line in request before a shared bus processes a bus transaction. If CATS finds that a processor has that cache line, it predicts that the processor with the cache line will send a cache coherence event after it watches the first address to access. Therefore, CATS transforms the bus transaction to notify the arrival of the cache coherence event.

```
1 : next_arbitration = bus_clock_period
2 : while (true) {
3 :    advance clocks of processors until next_arbitration
4 :    transaction = perform a bus arbitration
5 :    if (transaction is null) {
6 :       next_arbitration += bus_clock_period
7 :       continue;
8 :    }
9 :    cache_line = find a cache line for all other processors
10:    if (cache_line is not null) transform a transaction
11:    if (transaction unlocks the bus) set the unlocked flag
12:    else clear the unlocked flag
13:    process a bus transaction with the locked flag
14:    next_arbitration = acquire bus's next arbitration time
15: }
```

Figure 3. A pseudo code for a simulation kernel

Figure 3 shows a pseudo code for a simulation kernel which implements the proposed technique. The simulation kernel first predicts events at line 9~12 and advances the clock of the shared bus at line 13. After acquiring the next arbitration time at line 14, the kernel advances clocks of processors until the next arbitration time at line 3 and find the next transaction at line 4. In this way, the transaction in the shared bus drives executions of processors. That's why we call this technique as CATS.

To handle bus request events, the kernel checks a control signal of a bus transaction in line 12 and determines granularity of the shared bus. For cache coherence protocol, line 9 checks if any processor has a cache line which the current transaction accesses. If the cache line is founded, the kernel transforms the transaction at line 10.

3.2 Handling Events to Processors

We can model behavior of the shared bus cycle accurately due to techniques in section 3.1, which also means that we can have accurate timings of outgoing events. Based on this property, we revise a behavior model of the shared bus to store timings of occurred events and pack them into a reply to a bus transaction. In other word, the model only abstracts behavior, not timings of events. Because the simulation kernel shown in Figure 3 advances the clock of the shared bus prior than those of processors, processors acquire exact arrival times of incoming events with the captured timings.

```
1: slave = find_slave(transaction's address)
2: if (unlocked flag is set) access_num = 1;
3: else access_num=transaction's bytes to access / bus width
4: while (access_num>0) {
5:    bus's next arbitration+=acquire an access delay of the slave
6:    store bus's next arbitration into a reply to the transaction
7:    access_num--;
8: }
```

Figure 4. A simplified pseudo code for abstracted behavior model of the shared bus

Figure 4 shows a pseudo code for abstracted behavior model of the shared bus. Details of a bus protocol are omitted to simplify the explanation. First, the behavior model selects a slave for the bus transaction at line 1. If the transaction is related to cache coherence and transformed at line 10 in Figure 3, the processor with the cache line becomes a slave. In normal cases, a memory becomes a slave. Second, the model determines the number of accesses for the shared bus. If the transaction performs an unlocked transaction which is flagged at line 11 in Figure 3, the behavior model is set to process one memory transaction at line 2. Otherwise, the behavior model is set to advance its clock until it finish the bus transaction at line 3. For each memory access, the behavior model acquires an access delay from the slave and updates the next arbitration time of the bus at line 5. Moreover, it stores the time into a reply at line 6, which are utilized for processors to know arrival times of incoming events.

When a processor receives the reply from the shared bus, the processor utilizes the timings in different ways based on event types. The events can be interrupt events from other processors, events related to cache coherence and data events from the shared bus.

First, interrupt events from other processors are delivered through a shared bus. The bus can be a dedicated bus or shared by instruction and data accesses. For simplicity, in this paper, we assume that interrupt events share a bus with other accesses. Thus, when a processor sends an interrupt event to other processor, the event is delivered through the shared bus to an interrupt handler. Then the interrupt handler sends an interrupt signal to a destination processor.

Therefore, if a processor watches transactions to interrupt handlers on the shared bus, the processor knows when an interrupt event arrives or it does not. If the processor does not receive an interrupt signal, the processor advances its clock until the current transaction ends. Otherwise, the processor receives the interrupt signal when a static delay is passed after a data is written to the handler.

Second, when a processor writes data to a shared cache line, processors with the cache line invalidate or update the cache line as soon as the processors watch the first address at the shared bus.

Finally, if a processor supports an early resuming of pipeline stages, the processor does not need to suspend its pipeline stages at the end of the bus transaction. Instead, the processor resumes its pipeline stages with the first data of the cache line.

4 OS Sensitive Scheduling

In section 3, we showed that CATS reduces the number of instance switches with larger granularity, simulation time for a shared bus with abstraction and event scheduling overhead with static scheduling. However, we have to perform cycle accurate simulation for processor simulators to maintain cycle accuracy. Thus processor simulators have become a bottleneck of simulation performance.

When we analyze applications using multiple processors, we notice that interactions between tasks produce idle cycles of processors. Those idle cycles take larger portion of simulation time as the number of processors is increased. If we can catch when idle cycles of a processor happen and when a processor moves from idle cycles to active cycles, we can reduce simulation time consumed for processors by skipping idle cycles of processors.

Idle cycles of a processor occur when there is no active task available in the processor. In that case, an operating system switches its program context to an idle task which does nothing and has the lowest priority. To wake up from the idle task, an interrupt from other processor or a timer interrupt has to notify the processor. Therefore, we can ignore all events to the processor which executes the idle task until any interrupt event arrives.

In OS sensitive scheduling, we annotate a special code in the idle task. The code notifies the processor simulator to change its state to an idle state. When we advance the clock of the simulator, we do not execute behavior of the processor simulator in the idle state but update the local clock of the simulator. If we watch that any interrupt is delivered to the processor in the idle state, we change the state of the processor simulator to an active state and restart to advance the clock of the processor. However, even in the idle state, we update cache coherence of the processor to maintain cycle accuracy.

5 CATS Framework

We have implemented the proposed technique in a CATS framework which is based on SimpleScalar. The CATS framework configures different architecture platforms by using an architecture description script and provides programming interfaces for a multithreaded program model.

In an architecture description script, we specify configurations for processor models, an inter-connection network and an operating system. Currently we support ARM processors and an AMBA AHB bus [12] as an inter-connection network. Each processor defines a clock period and memory map entries which translate virtual memory in the processor to physical memory in a memory. If we define the same memory map entries for all processors, processors share the same program image. Otherwise, each processor will execute different images. A definition for an AMBA AHB bus includes a clock period, a bus width, connected memories and connected processors. In addition, we are working to support more processor models like PISA, Alpha and PowerPC and different inter-connections like multiple buses with bridges and network on a chip.

We implement an operating system as simulation library. Each simulator catches system calls to an operating system and emulates behavior of the operating system. An operating system can be easily configured by using a scheduling policy, a maximum number of tasks, an interrupt tick period, a flag to choose a single program image or multiple program images and involved processors.

6 Experiments

In experiments, we use Fedora Linux core 3 with Pentium-M 1.7Ghz CPU. The simulation framework is compiled by gcc 2.94.5 with –O3 option and applications are by arm-linux-gcc 2.95.2 with –O3 option. For simulations, we assume that processors and a memory are connected to a shared bus. Each processor has separated L1 caches with 8 KB each and a unified L2 cache with 64KB.

First we evaluate efficiency and accuracy of CATS compared to an original SimpleScalar ARM. We execute four applications of Splash-2 benchmark with a single processor on the CATS framework and the original SimpleScalar ARM. Four applications are fast Fourier transform (FFT), Cholesky factorization (CHOLESKY), radix sorting (RADIX) and volume rendering (VOLREND). In the experiments, memory delays are set to 18 processor cycles in non-sequential accesses and 2 processor cycles in sequential accesses.

Figure 5 first shows simulation efficiencies. We divide simulation times of CATS by those of SimpleScalar ARM which achieves 540K~700K instructions/s. Simulation performances of CATS are close to 80% of the SimpleScalar ARM, which means that CATS effectively reduces all components of simulation time except behavior model of processors. Second, errors in simulated cycles are under 0.003% except 0.13% of FFT when we extend a static delay model to an inter-connection network simulator applying CATS.

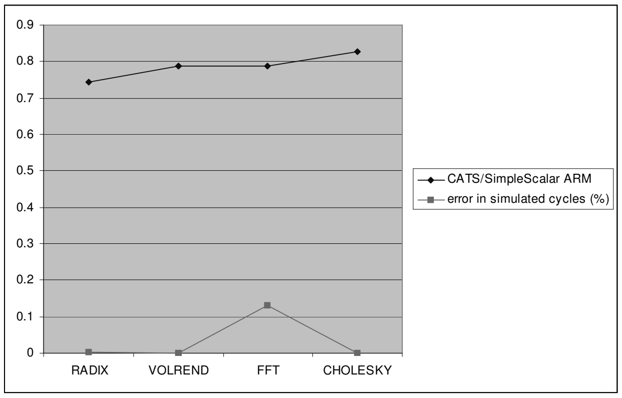

Figure 5. Simulation efficiency and accuracy of CATS compared to an original SimpleScalar ARM

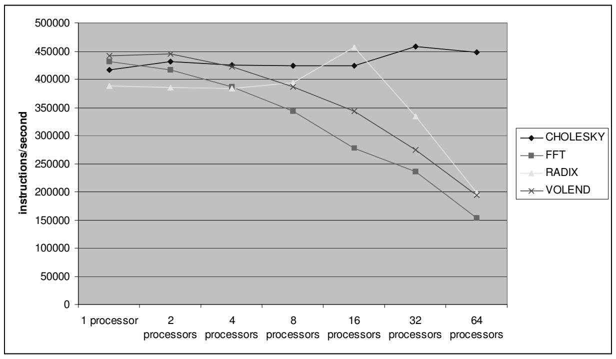

Figure 6. Simulation performances of four applications from Splash-2 benchmark

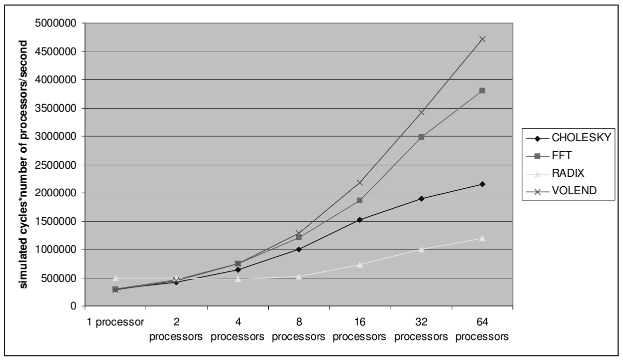

Figure 7. Simulation performances with simulated cycles*number of processors/second

Second, we change the number of processors from one to sixty four, and assume that a shared bus and processors have the same clock period and all memory delays are one processor cycle.

In Figure 6, we notice that changing number of processors does not affect simulation performances of CATS. However, delays from memory conflicts decrease cycles to process one instruction (IPC) and make processor simulators inefficient when we have more than 16 processors.

However, Figure 6 does not count instructions in idle cycles which are skipped by OS sensitive scheduling. Thus, in Figure 7, we calculate simulation performances by multiplying the number of processors to simulated cycles per second, i.e., processor cycles/s. Except RADIX which has idle cycles under 9% with 64 processors, all other applications have minimum 31% at 2 processors and 88% at 64 processors. Therefore, skipping idle cycles enables processor simulators to process more cycles per second.

7 Conclusion

We have shown that event predictions to a shared bus can enable us to advance the clock of the shared bus without interactions with processors. This is possible based on analysis of the processor simulators. The behavior model of the shared bus stores timings of events and provides cycle accuracy to processor simulators. Moreover, application behavior from an operating system makes it possible to skip idle cycles of processor simulators. All the proposed techniques are implemented in the CATS framework. The framework provides simulation performances comparable to TLM with cycle accuracy. As future work, we are working to provide additional processor and interconnection network models. We will examine accuracy with multiple processors by comparisons with a real multiprocessor system.

Acknowledgement

Our work was supported by a gift from Intel Corporation, a grant from UC Discovery program and IT scholarship program which is supervised by Institute for Information Technology Advancement & Ministry of Information and Communication, Republic of Korea. We are grateful to the anonymous reviewers for their insightful comments.

References

[1] CATS Framework, http://mesl.ucsd.edu/dhkim/CATS

[2] Doug Burger and Todd M. Austin, “The SimpleScalar Toolset, Version 2.0,” University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.

[3] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. “The SPLASH-2 Programs: Characterization and Methodological Considerations”. In Proceedings of the 22nd International Symposium on Computer Architecture, pp. 24-36, Italy, June 1995

[4] Benini L, Bertozzi D, Bogliolo A, Menichelli F, Olivieri M, “MPARM: Exploring the Multi-Processor SoC Design Space with SystemC”, The Journal of VLSI Signal Processing, Vol. 41, No. 2., Sep. 2005, pp. 169-182.

[5] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt, “Network-Oriented Full-System Simulation using M5”, Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads, Feb 2003.

[6] P. Schaumont, I. Verbauwhede, "Interactive Cosimulation with Partial Evaluation”, DATE 2004, Feb. 2004

[7] Chris Lennard, Davorin Mista, “Taking Design to the System Level”, the white paper for SoC Designer, a ARM Inc.

[8] Jin Yong Jung ; Sung Joo Yoo ; Kiyoung Choi, "Performance improvement of multi-processor systems cosimulation based on SW analysis ", DATE 2001, pp.749-753, Mar. 2001

[9] Dohyung Kim, Youngmin Yi and Soonhoi Ha, "Trace-Driven HW/SW Cosimulation Using Virtual Synchronization Technique", DAC 2005, Anaheim, June 13-17 2005

[10] L. Cai and D. Gajski, "Transaction level modeling: an overview", Proceedings of 1st international conference on Hardware/software codesign and system synthesis, Oct. 2003

[11] S. Pasricha, N. Dutt, and M. Ben-Romdhane, “Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration”, in Proc. Intl. Conf. on Design Automation, pp. 113-118, USA, Jun. 2004.

[12] “AMBA Specification Rev 2.0”, ARM Inc.