Fast Hardware-Software Coverification by Optimistic Execution of Real Processor

Sungjoo Yoo Jong-Eun Lee Jinyong Jung Kyungseok Rha Youngchul Cho Kiyoung Choi
Design Automation Laboratory
School of Electrical Engineering
Seoul National University
Seoul 151-742, Korea
{ysj,jelee,jyjung,contron,rams,kchoi}@poppy.snu.ac.kr

Abstract

To achieve fast verification of the software part of embedded system, we propose to run the target processor optimistically, which effectively reduces the synchronization overhead with other simulators. For the optimistic processor execution, we present a processor execution platform and state saving/restoration methods. We performed optimistic execution of ARM710A processor in the coverification of an IS-95 CDMA cellular phone system and obtained up to orders of magnitude higher performance compared with the case that the processor runs conservatively.

1. Introduction

Verification of system functionality and timing is one of the most difficult and important aspects of System on a Chip (SoC) design. For many system design teams, verification takes 50% to 80% of the overall system design effort [1].

For fast verification of the hardware part of the system being designed, cycle-based simulators and high performance emulation systems have been widely used. For the verification of the software part, various processor models such as register transfer models, instruction set architecture (ISA) models, and bus functional models are being used. Among them, currently the ISA model is widely used since it gives fast simulation performance and cycle-level accuracy (sometimes in cooperation with the bus functional model of the processor).

However, as the software part of SoC design gets more and more complex, verification of the complex software part is becoming a bottleneck to shortening time-to-market. As a method of fast verification of the software part, running it on the target processor itself seems to be the best solution if possible. In fact, there have been many processor evaluation systems developed by processor vendors or others [2].

From the viewpoint of coverification, however, such processor evaluation systems do not give sufficient coverification performance due to the high synchronization overhead with other simulators. Such overhead is common in the cases that hardware emulator execution and software simulator execution are involved in coverification.[1] In those cases, the overhead is reported to yield up to two orders of magnitude performance degradation [1].

To resolve the synchronization overhead problem between the real processor execution and the hardware part simulation[2], we apply optimistic simulation concept to the real processor execution since optimistic simulation reduces such overhead efficiently [3][4]. In performing optimistic execution of real processor, one of the crucial issues is how to implement state saving and restoration of the software part running on the processor. In this paper, we present a method of performing optimistic execution of a real processor including state saving and restoration. We performed optimistic execution of an ARM710A processor in the coverification of a CDMA cellular phone system based on the IS-95 specification [5][6].

This paper is organized as follows. In Section 2, we review related work. In Section 3, we describe our motivation for optimistic execution of real processor. In Section 4, we present a method of optimistic processor execution. Experimental results are given in Section 5. We conclude in Section 6.

[1] In this paper, **software simulator execution** means that a simulator, which can simulate the software part or the hardware part of the system being designed, runs on a simulation host.

[2] In this paper, **real processor execution** means that the target processor, for the software part of the system being designed, executes the software part. **Hardware (software) part simulation** means that a simulator simulates the hardware (software) part of the system being designed.

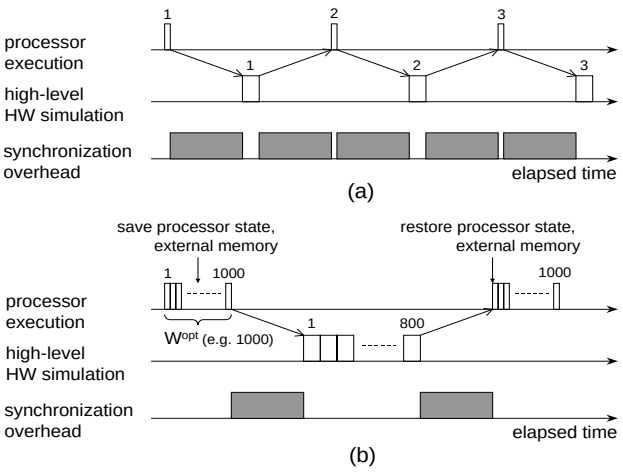


Figure 1. Synchronization overhead reduction by optimistic processor execution.

2. Related Work

As one example of commercial processor evaluation systems, the ARM Development Board (formerly known as PID7T) [7] from ARM Ltd. allows designers to run the prototype of application software on the ARM7TDMI processor. It supports JTAG-based in-circuit emulation functions. As one of general JTAG-based in-circuit emulation systems, winIDEA from iSystem GmbH [8] allows software debugging on various types of processors. To facilitate in-circuit debugging of embedded cores in SoC designs, an embedded in-circuit emulator synthesizer called ICEBERG [9] inserts and integrates the in-circuit emulation circuitry into a given RTL core of a microcontroller.

To reduce synchronization overhead in cosimulation, a concept called **hybrid synchronization** has been proposed in [3][4]. In hybrid synchronization, some simulator(s) run optimistically to reduce the number of null messages between the optimistic simulator(s) and the others. In [10], to reduce the state saving/restoration overhead in optimistic simulation, task-based state saving methods have been presented. For the reduction of message communication overhead in geographically distributed cosimulation, a concept called **hierarchically grouped message** has been presented in [11][12].

3. Motivation

In this section, we assume a case that the software part of the system is verified by running the real target processor and the hardware part (and the environment) is simulated by a high-level simulator, e.g. Ptolemy [13]. The case is common in the verification of the complex software part by extensive execution of the application software together with high-level hardware models having estimated or back-annotated hardware execution times.

In Figure 1, we assume that the processor execution and the hardware part simulation start to run concurrently at time 0. In the figure, blank rectangles and numbers on them represent respectively workloads and the corresponding local times in the processor execution and the hardware part simulation. Arrows represent messages, i.e. timestamped events, transferred between the processor and the simulator. Shaded rectangles represent **synchronization overhead** caused by the transfer of such messages.

In the coverification scenario shown in Figure 1 (a), both the processor and the simulator advance their local times by one system clock period in a lock step manner and exchange (null) messages **to detect the occurrence of events, especially, interrupts** transferred between the software and hardware parts. In such a coverification scenario, since (null) messages are inevitably exchanged between the processor and the simulator at every clock tick, the synchronization overhead caused by the transfer of messages can consume most of coverification runtime. Thus, we can hardly obtain enough performance improvement from real processor execution instead of software part simulation.

In hybrid synchronization [3][4], the high synchronization overhead can be reduced by performing optimistic processor execution as shown in Figure 1 (b). In the figure, we first run the processor optimistically for a time window of predetermined size $\mathrm{W^{opt}}$ (e.g. 1000). The processor stops after the time window $\mathrm{W^{opt}}$ elapses or at a time point $\mathrm{W^{opt}}$' ($< \mathrm{W^{opt}}$) when an event is sent to the simulator, sends a (null) message to the simulator, and waits for a message from the simulator. During the processor execution, **states of the software part** are stored at checkpoints in preparation for the potential **rollback**.

Then the simulator starts to run until the time point when the optimistic processor execution has stopped. It may stop earlier (e.g. at time 800) if it comes to send an event to the processor. In this case, since the timestamp of the message sent to the processor is earlier than the time point when the processor has stopped, **the processor execution rolls back to a checkpoint not later than the timestamp of the message**. If there is no message from the simulator to the processor, then the simulator stops at the time point $\mathrm{W^{opt}}$ (or $\mathrm{W^{opt}}$'). After determining a new $\mathrm{W^{opt}}$, the processor starts to run until $\mathbf{W^{opt}}$. Then, the coverification continues in this way. For further details on hybrid synchronization, refer to [3][4].

As shown in Figure 1 (b), by optimistic processor execution, we can reduce synchronization overhead **while preserving the timing accuracy of coverification**. To perform optimistic processor execution, we must support state saving/restoration of the software part and detection of software-to-hardware events. To do that, we define the state

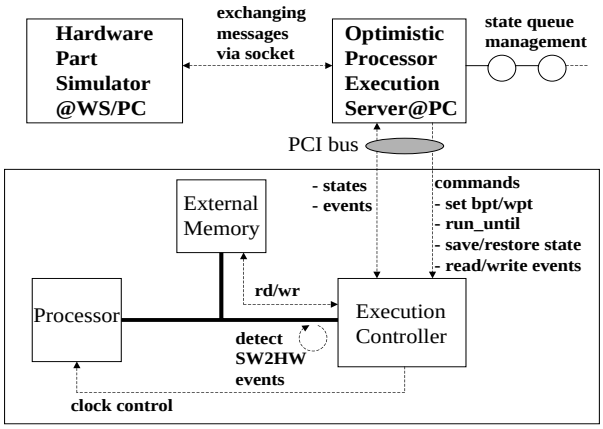


Figure 2. An optimistic processor execution platform.

of the software part running on the processor and propose a processor execution platform and state saving/restoration methods in the next section.

4. Optimistic Processor Execution

4.1. A Processor Execution Platform

For optimistic processor execution in hybrid synchronization, we use an optimistic processor execution server and a processor modeler as shown in Figure 2.

4.1.1 Optimistic Processor Execution Server

The optimistic processor execution server runs on a simulation host to which the processor modeler is connected.[3] It controls optimistic execution of the processor by issuing the processor execution (e.g. set breakpoints/watchpoints and run_until) and state saving/restoration commands to the processor modeler. It manages the saved states of the software part in its state queue. It exchanges messages with the hardware part simulator running on the same host or on a remote simulation host.

4.1.2 Processor Modeler

The processor modeler consists of a processor, external memory, and an execution controller. The execution controller receives commands from the optimistic processor execution server and performs processor execution control and saving/restoration of software states.

[3] In our implementation, the optimistic processor execution server runs on a PC to which the processor modeler (a PCB) is attached via PCI bus.

For processor execution control, the execution controller sets breakpoints/watchpoints on the external memory access from the processor and runs the processor until the breakpoints/watchpoints or for an amount of clock cycles (**run_until** function). The execution controller has a **clock counter** which registers the total number of clock counts of the processor. The execution controller saves (restores) the software state sending (receiving) it to (from) the optimistic processor execution server.

4.1.3 Detection of Software-to-Hardware Events

To detect events transferred to the hardware part (e.g. events on the address/data buses and control signal pins of the processor that occur during loading/storing data items from/to the hardware part), the optimistic processor execution server sets watchpoints to all the addresses of the data items transferred between the processor and the hardware part. On reaching one of the watchpoints, the execution controller stops processor execution, then sends the event to the optimistic processor execution server.

4.2. Definition of Software State

The state of the software part consists of two parts as follows.

- Processor internal state : the contents of registers (including condition code), cache, memory management unit (MMU), address/write buffers, etc.
- External memory state : the contents of readable/writable (RW) data area

The processor internal state is determined by the architecture of the processor used in the system being designed. In the case of ARM710A processor [14], the processor internal state consists of the contents of a total of 37 registers, 8 KB cache, MMU, and address/write buffers.

The external memory state is determined by the application software running on the processor. The RW data area is divided to global data area, heap area and stack area. The size and range of global data area are determined statically in the step of linking object codes into the executable image to be run on the processor. The size of heap area (used for dynamic memory allocation) varies during the application software run. In the case of stack area, although its size varies while the application software is running, it can be identified at any moment of software execution. In the case of ARM710A processor, since register R13 is used as the stack pointer, the stack area is determined by the contents.

4.3. State Saving and Restoration Methods

In this subsection, we present two state saving/restoration methods.

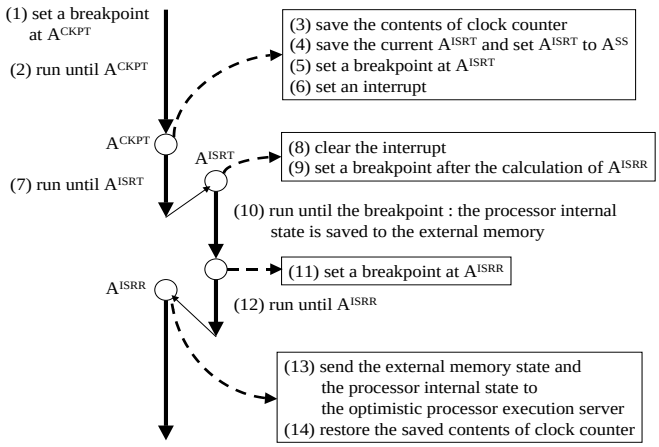


Figure 3. An ISR-based state saving method.

4.3.1 Interrupt Service Routine-based Method

In this method, we control the interrupt to the processor for state saving and restoration. On receiving an interrupt, the processor is assumed to jump to the interrupt service routine (ISR) target address $\mathrm{A}^{\mathrm{ISRT}}$ where the ISR starts. Since the application software also uses its own ISR's, this method requires the execution controller to take control of $\mathrm{A}^{\mathrm{ISRT}}$.

Figure 3 shows the state saving flow of the ISR-based method. To save the state of the software part at a checkpoint, the execution controller first sets a breakpoint at the checkpoint address $\mathrm{A}^{\mathrm{CKPT}}$ designated by the optimistic processor execution server and runs the processor until the breakpoint (step 1 and 2 in the figure). At $\mathrm{A}^{\mathrm{CKPT}}$, the execution controller first saves the contents of the clock counter (step 3) since the clock counter will count up during the execution of ISR. Then, it saves the contents of the current $\mathrm{A}^{\mathrm{ISRT}}$, i.e. the ISR target address of the application software and sets $\mathrm{A}^{\mathrm{ISRT}}$ to the start address of the subroutine (A^{SS}) that performs state saving (step 4). After setting a breakpoint at $\mathrm{A}^{\mathrm{ISRT}}$ (step 5), to force the processor to enter the ISR subroutine, the execution controller sends an interrupt to the processor (step 6).

At $\mathrm{A}^{\mathrm{ISRT}}$, i.e. A^{SS}, the execution controller clears the interrupt (step 8). The ISR return address $\mathrm{A}^{\mathrm{ISRR}}$, to which the ISR will return and where the application software continues to run, is calculated in the ISR.[4] The execution controller sets a breakpoint at an instruction which resides just after the code block of $\mathrm{A}^{\mathrm{ISRR}}$ calculation in the ISR (step 9). It runs the processor until the breakpoint. During the processor execution, the contents of the processor internal state (e.g. contents of registers) are saved into the external memory (step 10). At the breakpoint, the execution controller can set a new breakpoint at $\mathrm{A}^{\mathrm{ISRR}}$ (step 11). It runs the processor until the breakpoint (step 12).

At $\mathrm{A}^{\mathrm{ISRR}}$, the execution controller reads the external memory state and the processor internal state (previously saved at step 10) from the external memory and sends them to the optimistic processor execution server (step 13). As the last step, it restores the contents of the clock counter to the one saved in step 3. If $\mathrm{A}^{\mathrm{ISRR}}$ is different from $\mathrm{A}^{\mathrm{CKPT}}$, the number of clock cycles to be elapsed between $\mathrm{A}^{\mathrm{CKPT}}$ and $\mathrm{A}^{\mathrm{ISRR}}$ in the normal execution (without the ISR execution) is calculated and added to the contents of the clock counter.

The restoration of software state is performed in a similar way to the state saving. Comparing the state saving in Figure 3, in the state restoration, step 3 (in the figure) is not performed and the contents of the processor internal state and the external memory state are restored at step 10 and 13, respectively.

4.3.2 JTAG-based Method

For the processors which support in-circuit emulation functions based on JTAG boundary scan (IEEE Std. 1149.1 - 1990), the state of the software part can be saved and restored during the *debug state* where the processor internal state and the external memory state can be examined. In the debug state, the processor internal state is examined via a JTAG serial interface, which allows instructions to be serially inserted into the processor without using the external data bus. Thus, in the debug state, a sequence of single load/store instructions or block load/store instructions can be inserted to the processor and they will read/write the contents of the processor registers. These data can be serially shifted out via the JTAG serial interface.

In the JTAG-based method, the external memory state can be saved/restored by the execution controller as performed in the ISR-based method.

4.3.3 Limitation in the State Saving and Restoration

The accessibility to the processor internal state is determined by the processor architecture. For example, ARM7TDMI enables access to the internal state of MMU while ARM710A does not. But, neither of two processors enable access to the internal state of cache. Thus, the usefulness of optimistic processor execution depends on the accessibility to the internal states of target processors.

5. Experiment

5.1. Coverification Environment

We use a prototyping board [15] as the processor modeler. The board consists of an ARM710A processor, 2 MB

[4] In the cases of pipelined processors, $\mathrm{A}^{\mathrm{ISRR}}$ can be different from $\mathrm{A}^{\mathrm{CKPT}}$ depending on the instruction being executed when the processor fetches the instruction at $\mathrm{A}^{\mathrm{CKPT}}$.

SRAM, a XC4085 FPGA, and a PCI bus chip. It is connected to a PC (Pentium II, 500 MHz) where the optimistic simulation server runs. In our experiment, we run Ptolemy [13] on an UltraSparc I (143 MHz) as the high-level hardware simulator which does not perform optimistic simulation.[5] The optimistic simulation server and Ptolemy exchanges messages via Windows and Unix sockets.

We implement the simulation controller on the FPGA. We use the ISR-based method for state saving and restoration since ARM710A does not support JTAG-based emulation functions. Since the contents of cache and MMU can not be accessed in the ISR routine, we turned off the cache and MMU of ARM710A processor in our experiments. The simulation controller consumes 7% of CLB's in XC4085.

5.2. An IS-95 CDMA Cellular Phone System

We use a CDMA cellular phone system based on the IS-95 specification [5][6][16]. The IS-95 system consists of one mobile station, one base station, and air channel models. The mobile station consists of six CDMA modems (there are four channel types for the forward link[6] and two channel types for the reverse link[7]), a QCELP (Qualcomm Code Excited Linear Prediction) vocoder, and a call processor which performs system initialization, paging, call initiation, etc.

We model the QCELP vocoder in CGC (Code Generation in C) domain of Ptolemy [13] and generate C code for the software implementation. We model the call processor in Esterel and generate C code using Polis [17]. We run the call processor and QCELP vocoder of the mobile station on the ARM710A processor of the processor modeler and the other parts (mobile station modems, the base station, and air channel models) in Ptolemy.

We run a scenario of the IS-95 system run where the base station initiates a call to the mobile station and the two stations proceed a call initiation procedure exchanging paging, access, and forward traffic and reverse traffic channel messages. After the call initiation, the two stations start conversation.

Since the call processor and vocoder do not perform dynamic memory allocation, the heap area is not considered in the experiment. The executable image of the call processor and vocoder has size of 61.59 KB for code and read only data and 78.74 KB for readable/writable global data. The stack area is determined at runtime and saved (restored) by the state saving (restoring) ISR.

[5]The simulation on the Ptolemy side is not necessarily for hardware only.

[6]There are pilot channel, sync channel, paging channel, and forward traffic channel for the forward link from the base station to the mobile station.

[7]There are access channel and reverse traffic channel for the reverse link from the mobile station to the base station.

Table 1. Runtimes and the number of messages in the IS-95 system coverification

	cons. 710A	opt. 710A
No. messages	32,000,000	136,820
runtime (sec.)	12,066	53

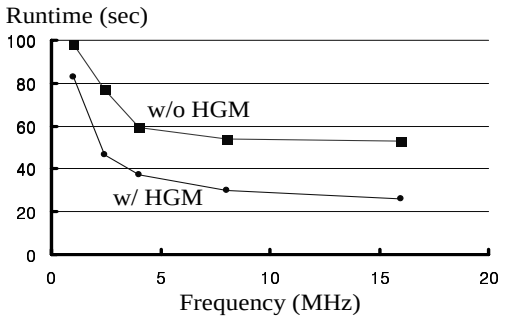
5.3. Results

Table 1 shows runtimes taken for the coverification of the IS-95 system. In the table, opt. (cons.) ARM710A represents the case that the ARM710A processor performs (does not perform) optimistic execution at 16 MHz clock frequency. The table shows that the optimistic processor execution enables 99.57% reduction in the number of messages transferred between the processor and Ptolemy as explained in Section 3. Such a significant reduction in the number of messages yields 228 times performance improvement compared to the runtime of the case that the optimistic processor execution is not used.

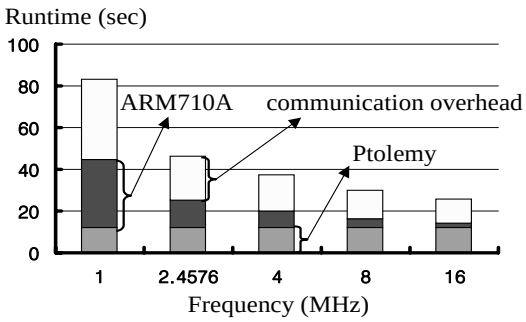
By analyzing the coverification runtime, it turned out that communication overhead between ARM710A processor and Ptolemy dominates coverification runtime. To analyze the effect of communication overhead, the runtimes were measured varying the clock frequency of the prototyping board. Figure 4 (a) shows the change of runtimes as the clock frequency increases. In the figure, HGM represents an optimization method called **hierarchically grouped message** which reduces synchronization overhead by reducing the number of event-carrying messages [12][11]. As shown in the figure, HGM concept gives about two times more performance improvement in our experiments of optimistic processor execution. As the frequency increases, the decrease of runtime saturates. Figure 4 (b) shows the reason. In the figure, as the frequency increases, the portion of ARM710A execution and communication overhead decreases. The communication overhead consists of two portions : (1) overhead of the optimistic processor execution server accessing the execution controller on the FPGA and (2) overhead of transferring messages between PC and workstation over LAN. As the clock frequency increases, the first portion in the communication overhead decreases. However, the other portion of communication overhead and the portion of Ptolemy run in total runtime does not change. Thus, the decrease of runtime saturates as the clock frequency reaches 16 MHz.[8]

Table 2 shows comparison of total coverification runtimes between the case that a commercial ARM simulator, ARMulator runs on the Ultra I workstation together with Ptolemy and the case that the ARM710A processor is used

[8]The ARM710A processor can run at maximum 25 MHz with 3.3 V supply voltage.



(a)



(b)

Figure 4. Runtime change v.s. clock frequency : (a) effect of HGM concept and (b) decomposition of runtimes w/ HGM.

Table 2. Comparison of total coverification runtimes

ARMulator		opt. 710A	
w/o HGM	w/ HGM	w/o HGM	w/ HGM
99	85	53	26

instead of ARMulator. Comparing the runtimes of ARMulator case and ARM710A case when HGM concept is applied, 3.27 ($=\frac{85}{26}$) times performance improvement could be obtained by running the real processor.

To investigate the overhead of the ISR-based state saving/restoration method, we measure the runtime of a single state saving/restoration by averaging total 10,000 runs of state saving/restoration of the IS-95 system. The state saving and restoration consume 61 msec and 38 msec, respectively.

6. Conclusion

In this paper, we present a method that reduces the synchronization overhead in coverification by performing optimistic processor execution. We obtained significant performance improvement by optimistic execution of ARM710A processor in the coverification of an IS-95 CDMA cellular phone system.

Our future work includes developing a new processor modeler that supports much faster state saving and restoration. For extensive application of optimistic processor execution, researches on efficient access to processor internal state such as cache, MMU, address/write buffers, etc. can be one of future directions.

References

[1] M. Keating and P. Bricaud, *Reuse Methodology Manual*, Kluwer Academic Publishers, 1999.

[2] ARM Ltd., "ARM Development Boards", *available at http://www.arm.com/DevSupp/Boards/*.

[3] S. Yoo and K. Choi, "Synchronization Overhead Reduction in Timed Cosimulation", *Proc. IEEE International High Level Design Validation and Test Workshop*, pp. 157–164, Nov. 1997.

[4] S. Yoo and K. Choi, "Optimizing Timed Cosimulation by Hybrid Synchronization", *to appear in Design Automation for Embedded Systems, Kluwer Academic Publishers*.

[5] TIA/EIA-95A, "Mobile Station-Base Station Compatibility Standard for Dual-Mode Wideband Spread Spectrum Cellular Systems", 1995.

[6] S. Yoo, J. Lee, J. Jung, K. Rha, Y. Cho, and K. Choi, "Fast Prototyping of an IS-95 CDMA Cellular Phone : a Case Study", *Proc. the 6th Conference of Asia Pacific Chip Design Languages*, Oct. 1999.

[7] ARM Ltd., "The ARM Development Board - ARM7TDMI Version", *available at http://www.arm.com/DevSupp/Boards/pid7t.html*.

[8] iSystem GmbH, "winIDEA", *available at http://www.isystem.com/Products/F_Products.htm*.

[9] I. Huang and T. Lu, "ICEBERG: An Embedded In-Circuit Emulator Synthesizer for Microcontrollers", *Proc. Design Automation Conf.*, pp. 580–585, June 1999.

[10] S. Yoo and K. Choi, "Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model", *Proc. Int. Workshop on Hardware-Software Codesign*, pp. 71–75, Mar. 1998.

[11] S. Yoo and K. Choi, "Optimizing Geographically Distributed Timed Cosimulation by Hierarchically Grouped Messages", *Proc. Int. Workshop on Hardware-Software Codesign*, pp. 100–104, May 1999.

[12] S. Yoo, K. Choi, and D. Ha, "Performance Improvement of Geographically Distributed Cosimulation by Hierarchically Grouped Messages", *to appear in IEEE Transactions on VLSI Systems*.

[13] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: a framework for simulating and prototyping heterogeneous systems", *Int. Journal of Computer Simulation, special issue on Simulation Software Development*, vol. 4, pp. 155–182, Apr. 1994.

[14] LG Semicon Co., Ltd., "32-bit-microprocessor GMS30C710A Data Book", *LGSARM DDI 0001A*, Apr. 1997.

[15] D. Lim, K. Na, and S. Yoo, "Design Automation Lab. Prototyping Board", *available at http://poppy.snu.ac.kr/Codesign/DAL_P98A/*.

[16] Qualcomm, Inc., "CDMA System Engineering Training Handbook", 1993.

[17] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems*, Kluwer Academic Publishers, 1997.