PAPER *Special Issue on VLSI Design and CAD algorithms*

Efficient and Flexible Cosimulation Environment for DSP Applications*

Wonyong Sung† *and* **Soonhoi Ha**†, *Nonmembers*

SUMMARY Hardware software codesign using various hardware and software implementation possibilities requires a cosimulation environment which has both flexibility and efficiency. In this paper, a hardware software cosimulation environment is developed using the backplane approach and optimized synchronization. To seamlessly integrate a new simulator, this paper defines and implements the backplane protocol for communication and synchronization between client simulators. Automatic interface generation facility is also devised for more effective cosimulation environment. To enhance the performance of cosimulation backplane, a series of optimized hardware software synchronization methods are introduced. Efforts are focused on reducing control packets between simulators as well as concurrent execution of simulators without roll-back. The environment is implemented based on Ptolemy and validated with a QAM example run on different configurations. With optimized synchronization method, we have achieved about 7 times speed-up compared with the lock-step synchronization.

key words: Hardware-Software Cosimulation, Cosimulation Backplane, Optimized Synchronization

1. Introduction

Cosimulation is the major tool to validate the model of heterogeneous reactive embedded systems, which usually consist of programmable components as well as the application specific hardware modules. Through cosimulation, we can validate the functional correctness of the hardware and software working together, ahead of the final synthesis step. Also, cosimulation enables us to evaluate each design decision such as partitioning and component selection[1].

Cosimulation has gained extensive research focuses but with diverse approaches depending on their different emphasis on conflicting goals such as cosimulation speed, accuracy, flexibility, and so on[2]. To make a flexible cosimulation environment, we use a heterogeneous approach rather than a unified one. In the unified approach, the entire system, both hardware and software, is described with a single specification model, usually a hardware description language such as VHDL. The unified simulation is simple but computationally inefficient and not flexible. In a heterogeneous approach, component simulators are separate processes running concurrently and cooperatively. We propose a backplane approach, in which a new simulator has only to define the interface to the backplane, leaving existent simulators unchanged. Then, we may employ different models of component simulators with a tradeoff between accuracy and performance. Moreover, we can perform distributed cosimulation, in which component simulators run on different machines.

In a heterogeneous approach, component simulators communicate with each other. The previous approaches usually adopt direct connection of hardware simulators and software simulators(or host-compiled processes) [3][4][5]. They have a major drawback that a new simulator can not be added without cost of redefining the interfaces of previous existent simulators. Defining interface, however, is a tedious and error prone work. Moreover, in each design iteration, the partition is remade and the interface code should be rewritten. Our work also presents an automatic interface generation to reduce the user's burden[6].

As system complexity grows, cosimulation speed becomes a great concern. Research efforts on cosimulation speedup include using hardware accelerator, changing cosimulation models across levels of abstractions, and reducing the cosimulation overheads as tried in this paper. Since the proposed backplane processes the messages between component simulators with event driven scheduling mechanism, timed cosimulation can be performed. The time is synchronized when the hardware and software simulators exchange messages[5]. To reduce the cosimulation overhead, one may use the optimistic approach in which a component simulator rolls back when it receives a past event from the other simulator. This approach assumes that the component simulator supports rollback mechanism that is usually not the case. In this paper, we propose another approach to optimize the conservative timed cosimulation. In case the application task graph is tree-structured, we obtain a significant gain of cosimulation speed.

This paper is organized as follows: Section 2 reviews our codesign workflow. Section 3 describes the concept and implementation details of the proposed cosimulation backplane with the automatic generation facility. Section 4 presents methods to optimize the cosimulation performance and section 5 shows an example and experimental results. Finally, we conclude by summerizing our main contribution.

†The authors are with the Department of Computer Engineering in Seoul National University

*Parts of this paper were presented at ASPDAC'98 and DATE'98 conferences. This work was supported by KOSEF(971-0906-040-2).

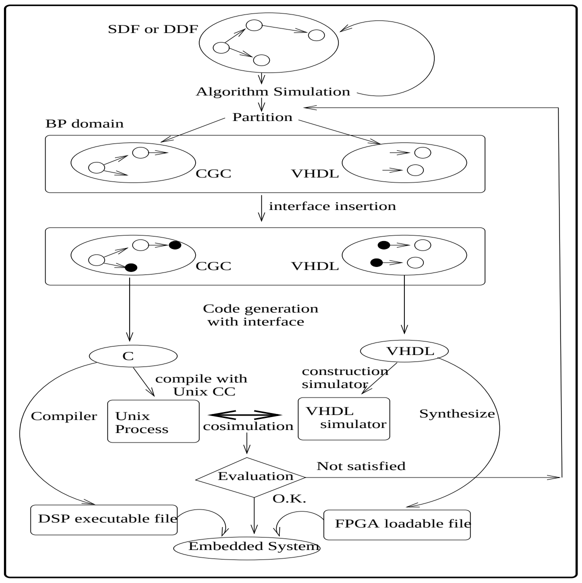

Fig. 1 Hardware Software Codesign Workflow

2. Codesign workflow

The codesign workflow consists of many stages from system specification to system synthesis. The codesign process presented in this paper is shown in figure 1.

A dataflow graph is chosen as an initial specification for a given application and simulated for algorithm verification. In a dataflow graph, a node represents a functional module and an arc represents the flow of data which also indicates the dependency between nodes. A node can be executed only when all its input arcs contain data samples produced from the source node. Dataflow models are effective for representing most types of DSP applications[7].

The next step is to partition the initial dataflow graph into two kinds of subgraphs, software graphs and hardware graphs. After partitioning, each subgraph is modified in order to add interface nodes at the graph boundary. From partitioned graphs, C and VHDL codes are generated including interface code. The generated C codes are compiled into UNIX processes and the generated VHDL codes from the hardware graph are passed to the VHDL simulator for hardware simulation. Therefore, several UNIX processes are running concurrently and cooperatively: C processes and VHDL simulators.

With simulation results, the evaluation module makes a decision whether the current partition satisfies the system requirements. Unless they are satisfied, the codesign process continues to iterate from the partition stage to the evaluation stage. Otherwise, the software executables and hardware modules are synthesized through a compiler and a behavioral synthesis tool, respectively. Of the whole codesign procedure in figure 1, this paper covers the topics related with cosimulation within the dashed line box.

3. Cosimulation Backplane

3.1 Cosimulation configurations in previous works

To build a cosimulation environment is to combine heterogeneous simulators. For hardware simulation, most people use a hardware simulator such as a VHDL or Verilog simulator. For software simulation, however, there have been three approaches pursued with their own merits and limitations.

First, a software program is executed with a processor simulator connected to a hardware simulator, which may achieve the most exact timing estimation[8]. For large systems, however, this approach is prohibitive because of its extremely long simulation time. Second, a software program is run as a module of the hardware simulator via its foreign interface [9]. In this approach, software processes, which are fully controlled by the VHDL simulator, are not adequate to model separate concurrent processes with the hardware.

Finally, a software program is executed on the development processor and communicates with the hardware simulator through UNIX IPC. In [3], the hardware and software descriptions are treated as separate UNIX processes which communicate through BSD sockets. Since it is targeting a specific application, it uses a fixed communication model between the hardware and software. A similar approach in [4] maintains two separate descriptions for hardware and software through the entire codesign process and models the communication as a message passing system. These works, however, consider only a fixed combination of simulators and a fixed topology of connections.

In all aforementioned approaches, there is no standard interface defined so that new communication code needs to be defined to integrate a new simulator. Also, data exchange between the software and the hardware parts is hidden in the simulated program. So, it is not visible to be probed. For arbitrary connection of various simulators, [10] defines a standard interface through which each simulator can communicate with others. In their approach, they use a C process as a standard interface so that a simulator should send the data to an automatically synthesized C process from which the destination simulator receives it. Their approach supports only dataflow model of computation, which implies timed cosimulation is not possible. Using a standard interface between heterogeneous simulators is the main feature of the Ptolemy[11], their cosimulation platform. Even though our research is also based on the Ptolemy environment, we propose another approach: backplane approach.

In our backplane approach, software simulators(processes) are run on the development processor and communicate with the backplane. A hardware simulator also communicates with the backplane. Therefore, the backplane is the master process to manage

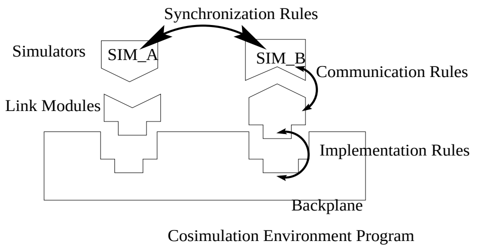

Fig. 2 The architecture of cosimulation backplane

the interprocess communication between software processes and the hardware simulators. As a standard interface for cosimulation, the backplane defines several rules that the client processes should meet. A new simulator can be seamlessly integrated with the backplane if it satisfies the interface rules.

A study on automatic interface synthesis for cosimulation is found in [12], where the VCI tool is developed in order to generate VHDL entities from the interface description called VCI specification. However, a designer should make the I/O descriptions whenever the graph topology is modified. A more automated approach of interface generation is studied in [10]. In their work, however, the application specification semantics are limited to a specific class of dataflow known as SDF[7]. Within the limitation of SDF semantics, they are able to statically schedule the graph and guarantee a deadlock free execution.

We aim to develop a technique of automatic interface generation from the partitioned dataflow graph. As shown in figure 1, the automatic interface generation is simply realized by insertion of communication nodes and generation of C or VHDL codes from them. Communication nodes are predefined in the library. [6] describes implementation details of automatically generated interface for VHDL simulation. The generated interface is compliant to the backplane protocol which is described in the following section.

3.2 Cosimulation Backplane

Figure 2 shows an architecture of the simulation backplane environment designed and implemented in this paper. The cosimulation environment is composed of three parts: the backplane program, interface link modules (one for each simulator) and the client simulators. To integrate a simulator, one link module needs to be created and placed between the simulator and the backplane. It acts as an interpreter between the simulator specific communication protocol and the backplane protocol. When a simulator runs with the generated code, the need for interprocess communication between the client simulator and the backplane arises. The backplane protocol is a standard way of communication between them. It consists of three groups of rules. They are communication rules, synchronization rules, and implementation rules.

Communication Rules This group of rules determines how the simulator and the backplane establish, maintain, and terminate the connection. Since the client simulator is invoked by the backplane, the backplane takes the initiative for the establishment of connection by calling a function defined in the link module for the simulator. Since we use the Berkeley socket IPC mechanism, the client process may run on a different machine to make a distributed cosimulation. The backplane is the scheduling engine of event driven model of computation. During cosimulation, the exchange of packets is serialized by the backplane to ensure the execution order of communication among multiple simulators. To terminate the connection, a special control packet is used. By receiving the termination packet generated by the backplane, a link module terminates the connection. When the client module terminates the simulation by itself, the simulator should notify the backplane by the sending a termination packet.

Synchronization Rules A group of rules is required for synchronization between client simulators. Though there is one physical socket connection between the backplane and a client process, there may be multiple logical connections in case there are more than one arcs at the partitioned boundary. Therefore, each packet needs to be identified from which port it is sent or to which port it is delivered. We assign a unique identification number to each port of the partitioned graph. When the backplane sends a packet to the client process, this *id* number is delivered as a part of the packet header. Since the backplane also inserts a time stamp into a header, a packet from the backplane contains three fields: *id, time stamp, and message(data).*

We enforce that the client process sends a signaling message, DONE signal packet, even though it does not generate a valid message at the current execution. In case multiple messages are transferred from/to the same port, a GO signal packet or a DONE signal packet is appended on each sequence of packets. By receiving this signal packet, the client simulator and the backplane can detect the end of transmission. These control packets are associated with negative id numbers.

The difficulty of synchronization among multiple concurrent event driven simulators is well understood[13]. If simulators use different time scales, the time scale is changed when packing and unpacking the message. By exchanging the time information, each simulator is controlled not to make a causality error, which is a main issue of section 4.

Implementation Rules This group of rules defines how to implement the link module which plays a role of a driver for its simulator. Like a conventional device driver in an operating system, its internal implementation is dependent on the supporting simulator while the backplane calls the same function. There are five call-back functions defined in the current implementation as listed in table 1.

In the setup stage, the backplane calls "modifyGalaxy" and "setPTInterface" functions once for each link module. In the "modifyGalaxy" function, the link module modifies the partitioned graph by inserting the communication nodes. The "setPTInterface" function makes a socket code with TCP port number given as an input parameter. After the C and VHDL codes are generated, the client simulators are invoked. At the beginning of cosimulation the client simulator calls "hostConnect" function to establish a socket connection with the backplane. In fact, a socket connection is established between the link module and the simulator.

Before the simulation of VHDL subgraph, the VHDL entities defined as communication nodes are inserted by the "modifyGalaxy" function. Therefore, any VHDL simulator supporting a foreign interface can be added without any modification of the program source, which is an important requirement for a backplane to support a commercial simulator like Synopsys VSS simulator. Sending and receiving messages from/to the backplane is done by the "ReadFromDevice" and "WriteToDevice" functions, respectively. The internals of the functions are dependent on the link module implementation. How the automatically inserted entities work can be found in [6].

4. Optimized Cosimulation

This section describes a series of methods to enhance cosimulatin performance, while not loosing the benefits of our cosimulation backplane. There are two considerations from which the optimization mode of the cosimulation is selected before the cosimulation is performed. One is the topology of the partitioned graph and the other is whether the client simulator discloses its scheduling information or not.

Basic Conservative Timed Cosimulation: When the backplane sends a message to the client simulator, it appends the timestamp to indicate the current global time. The client simulator executes until its local clock reaches the global current time and sends the response packet. Even when there is no response packet at the current global clock, the client simulator sends a dummy response packet to notify the end of the time advancement and the local next event time. On receiving the response packet, the backplane schedules this response packet to its event queue.

Since not all the event driven simulators allow the user to get the next event time, the local next time is assumed a unit time increment from the current local time. This is the basic lock-step synchronization mode and is depicted in figure 3(a).

Table 1 Call-back Functions

modifyGalaxy	add communication nodes
setPTInterface	create socket connection code
hostConnect	make a socket connection
ReadFromDevice	send a message
WriteToDevice	receive a message

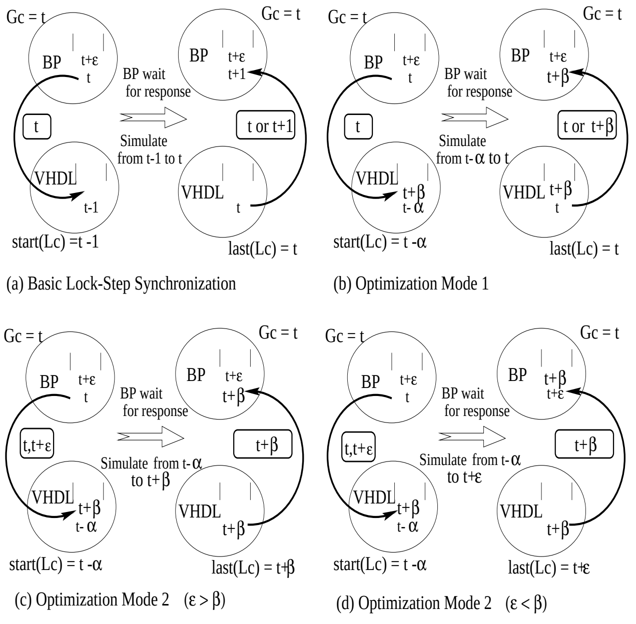

Fig. 3 Synchronization schemes in various optimization modes

While this scheme always guarantees timing correctness, it will be very inefficient especially when the event interval of the client simulator is much shorter than that of the backplane. Since the local clock of the client simulator can not advance without the backplane's intervention, at least two messages should be exchanged at each clock advancement between the backplane and the client simulator.

Optimization Mode 1 : The simple but effective optimization is the use of the next event time to reduce the synchronization points of cosimulation. If the client simulator has an API to get the next event time, the client simulator requests to the backplane to schedule itself at the next event time not after a unit-time increment. For example, Synopsys VSS VHDL simulator provides **getNextEventTime()** in the CLI library. As shown in figure 3(b), it reduces the synchronization points drastically especially when the time unit of the client simulator is much smaller than the event interval. In basic and optimization mode 1, following inequality(1) is always true, which means the local clock is never ahead of the global clock.

(1) $G_c = last(L_c) \geq start(L_c)$

Optimization Mode 2 : In mode 2, we allow the local clock to be ahead of the global clock while inequality (2) and (3) is true. The two inequalities mean that the client simulator is confirmed not to receive any past event.

(2) $G_c \geq start(L_c)$

(3) $t + \epsilon = next(G_c) \geq last(Lc)$

As shown in figure 3(c) and (d), if there is only one client simulator, the backplane sends the global next event time, which becomes the stop time of the

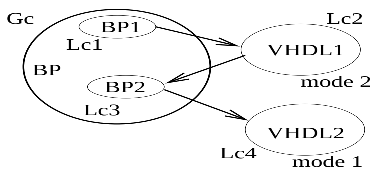

Fig. 4 Mode3 for tree structured graphs

client simulator, with input messages. Then, the client simulator can proceed until its local clock reaches the given stop time (case in (d)) or it generates a response message to the backplane(case in (c)). If it generates a response message before the stop time as in figure 3(c), the message becomes the next event in the backplane. Otherwise, the client simulator reaches the global next event time without producing result data as shown in figure 3(d). Then, the client simulator sends a response packet which contains the local next event time when the backplane will reschedule the client simulator.

Optimization mode 3 : Up to mode 2, the backplane waits until the client simulator responds after it sends input messages. Therefore, no parallelism is exploited in the distributed cosimulation. In mode 3, we allow to send the simultaneous messages to multiple client simulators at the current global time. The global clock of the backplane, however, can not advance until it receives response packets from the client simulators. We run client simulators concurrently in the distributed environment.

The more optimization is possible when the partitioned graph is tree-structured as shown in figure 4. Here, the inequality (1), (2), and (3) are not forced to meet while inequality (4) and (5) are met.

(4) $L_{c2} \geqq L_{c3} \geqq L_{c4}$

(5) $for\ each\ i, next(L_{ci}) \geqq L_{ci}$

We topologically sort the partitioned graph : backplane, VHDL1, backplane, VHDL2 in figure 2. Then, the ancestor simulator can be ahead of the descendant simulator. Therefore, the backplane keeps track of an event time for each arc separately, not for the whole event queue. The backplane allows a past event to the global queue as long as it is not a past event on the associated arc. In our experiments, only the top-most client simulator run in mode 2. The other simulators run in mode 1 because it is not guaranteed that it will not receive any past event from the ancestor simulator after local clock advancement. While other optimization ideas are also found in previous works, we believe that this optimization based on the topological sort is unique in this paper.

5. Experiments

A 16-QAM modulation is chosen to demonstrate the proposed cosimulation environment. The QAM16 modulator produces a 16-point quadrature amplitude modulated signal. To make a modulated signal wave, a

Table 2 Runtime performance of cosimulation in various timed cosimulation mode

Mode	Total(msec.)	BP	HW	SW	IPC
Basic	2,105,881	13.7%	4.2%	0.1%	82%
Mode1	473,277	39.8%	0.9%	0.4%	58.9%
Mode2	308,643	75.1%	1%	0.7%	23.2%

Table 3 Number of Control/Data Packets(unit : packet)

Mode	Total	Data	Control
Basic	667,047	3,842	663,205
Mode1	24,805	3,842	20,963
Mode2	21,755	3,842	17,913
Mode3	29,801	10,535	19,266

Table 4 Comparison cosimulation time between serialized and parallel execution in mode 3 (unit : msec.)

	Total	HW1	HW2	SW	IPC
Serial	934,169	7,515	4,076	2,088	411,672
Parallel	586,772	7,417	4,212	2,048	403,998

raised cosine pulse is used, where the excess bandwidth is 100% and the carrier frequency is twice the symbol rate. At this state, we manually partitioned the graph into software and hardware aiming to validate the correctness of the proposed cosimulation method.

The results shown in table 2, 3 and 4 are cosimulation time and overhead obtained by reading system timer at certain time points using gettimeofday() UNIX system call. We define the cosimulation time into four parts: VHDL simulation time, software simulation time, cosimulation backplane time, and transmission time (IPC : InterProcess Communication). After we measure the other times, we calculate the backplane time by subtracting them from the total time. Profiling results in table 2 and 4 are gathered after we cosimulate the QAM example with 320 loop counts. Under assumption that the transmission bandwidth is 19,200bps, the 320 loop counts mean 1 second of real time. The reason why the VHDL simulation time is much smaller than the total cosimulation time is that the VHDL model is very simple. The main source of low performance of cosimulation is the IPC overhead. The IPC overhead is mainly due to the number of control packets between the backplane and the client simulator. Table 3 shows how many control packets are needed between the backplane and the VHDL simulator. The IPC time as well as the number of control packets is reduced using mode 1 and 2. The backplane part, which includes the global event queue management, is also reduced according to the reduction of the number of control packets.

To experiment the optimization mode 3, we partition the hardware module into two parts. Each submodule is simulated on the different machine. Even though more computational resources are used, as shown in table 3, the QAM simulation in mode 3 shows worse performance than mode 2 because the number of packets are increased due to the partition of the VHDL module. The other reason is that only one of the two VHDL simulators uses the mode 2. Since the optimization approach as well as the partitioned graph in mode 3 are quite different, comparison of mode 3 with other modes is not meaningful. Thus, we compare the result in case of serialized run of simulators with that in case of parallel run.

6. Conclusion

In this paper, we have described a backplane approach of cosimulation. Using the backplane protocols, defined and implemented in this paper, we can achieve a cosimulation environment which integrates a new simulator seamlessly. Only a link module is needed to be created for integration and no modification of simulator source code is required. We assumed that the input specification is a coarse-grain dataflow graph which is partitioned into software and hardware subgraphs. By adding communication nodes to the partitioned subgraphs, we achieve automatic interface generation.

Within our cosimulation environment, various cosimulation speedup approaches are devised and implemented. The efforts are divided into two categories: shrinking the number of control packets and utilizing concurrent simulation. Our efficient yet flexible cosimulation environment will be an useful validation and evaluation tool in the core-based design era.

The proposed approaches have been implemented based on Ptolemy, which provides our system with strong facilities such as graphical user interface, visualization, and C/VHDL code generation from the dataflow graph. To validate our cosimulation environment, we showed a QAM example which runs on two different configurations of cosimulation.

References

[1] C. Passerone, et. al., "Fast and Accurate Hardware-Software Co-simulation Using Software Timing Estimates", CODES/CASHE' 96, 1996.

[2] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Methods, Validation, and Synthesis", Proceedings of the IEEE, Vol. 85, No. 3,March 1997.

[3] D. Becker, R. Singh, and S. Tell, "An Engineering Environment for Hardware/Software Cosimulation", Proc. Design Automation Conf. ACM, 1992, pp. 129-134

[4] Donald E. Thomas, J. K. Adams, H. Schmit, "A Model and Methodology for Hardware-Software Codesign", IEEE Design and Test of Computers, pp.16-28,September 1993.

[5] K. ten Hagen and H. Meyer, "Timed and Untimed Hardware/Software Co-simulation : Application and Efficient Implementation", 2nd. International Workshop on Hardware Software Codesign, Cambridge, Massachusetts, Oct. 7-8, 1993

[6] Wonyong Sung, Moonwook Oh and Soonhoi Ha, "Interface Design of VHDL Simulation for Hardware-Software Cosimulation", Proceedings of APCHDL'97, 1997

[7] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow", proceedings of the IEEE, vol. 75, no. 9, pp. 1235-1245, 1987

[8] R. Gupta, C. Coelho, and G. Demicheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software components", 29th DAC, pp.225-230, 1992

[9] J. P. Soninen, et. al. , "Co-simulation of Real-Time Control Systems", IEEE/ACM Proc. of Euro-Dac'95, pp. 170-175, 1995

[10] J. Pino, Michael C. Williamson and Edward A. Lee, "Interface Synthesis in Heterogeneous System-Level DSP Design Tools", IEEE International Conference on Acoustics, Speech, and Signal Processing, 1996.

[11] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschimitt, "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, April, 1994.

[12] C. A. Valderrama, et. al. , "A Unified Model of Cosimulation and Co-Synthesis of Mixed Hardware/Software Systems", European Design and Test Conference, 1995

[13] Jayadev Misra, "Distributed Event-Driven Simulation", ACM Computing Surveys, Vol. 18,no. 1,pp. 39-65, Mar., 1986.

Wonyong Sung received his B.S. and M.S. in computer engineering from Seoul National University, Korea, in 1990 and 1992, respectively. From 1992 to 1994, he was a member of technical staff at Nanum Tech. Inc. in Seoul, Korea. Currently, he is working towords Ph.D degree at the same university. His research interest is hardware software codesign and cosimulation.

Soonhoi Ha is an assistant professor in the Computer Engineering Department at Seoul National University, Seoul, Korea. His research interests include parallel processing, design methodology for digital systems, and hardware-software codesign. He has been actively involved in the Ptolemy project since its birth. He received the B.S. and the M.S. degrees in electronics from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively. He received the Ph.D. degree in the Electrical Engineering and Computer Science Department at U.C. Berkeley in 1992. He is a member of ACM and the IEEE Computer Society.