Performance Improvement of Multi-Processor Systems Cosimulation based on SW Analysis

Jinyong Jung† Sungjoo Yoo‡ Kiyoung Choi†

†Design Automation Lab
Seoul National Univ., Korea

‡SLS Group
TIMA/INPG, France

Abstract

In this paper, we propose a method for performance improvement of multi-processor systems cosimulation by reducing synchronization overhead between multiple simulators. To reduce the amount of simulator synchronization, we predict synchronization time points based on a static analysis of application software running on each processor. In the experiments with real embedded systems, we obtained up to orders of magnitude higher performance in cosimulation runtimes.

1 Introduction

One of crucial problems that system designers currently face is validating complex system software (SW) on a system-on-chip (SoC). The difficulty comes from the fact that (1) the SW portion of complex embedded systems begins to dominate system functionality and resources and (2) multi-processor architectures are getting more and more popular [1][2]. To validate the SW portion of an embedded system, in current design practice, cycle-accurate cosimulation with instruction set simulators (ISS's) is widely adopted.

In cycle-accurate cosimulation of multi-processor embedded systems, one of crucial problems is to reduce **synchronization overhead**. It results from the fact that multiple ISS's are involved in the simulation and they should synchronize with each other to achieve cycle accuracy. Especially when **interrupt**s are used for the processor to communicate with other processors or other HW modules [3], which is popular in current multi-processor architectures, the problem gets more severe. In such a case, each ISS should exchange messages with other simulators at every instruction execution to detect the occurrence of an interrupt sent to the processor that the ISS is simulating. Such exchange of messages is executed via time-consuming inter-process communication such as Unix socket or shared memory [4]. Thus, the overhead of message exchange can cause a significant degradation in the performance of cycle-accurate cosimulation.

To reduce the synchronization overhead in cycle-accurate cosimulation of multi-processor embedded systems, we propose a cosimulation method based on the analysis of application SW. The proposed method achieves the reduction from the prediction of the time point when inter-process communication occurs thereby reducing redundant message exchanges (that are required only to detect the occurrence of interrupt) between simulators. The advantage of the proposed method is improving the cosimulation performance with negligent analysis overhead while preserving the simulation accuracy. Since the proposed method is based on the analysis of application SW, it is hard to be applied to HW simulators.

This paper is organized as follows. We review related work in Section 2. We explain the synchronization overhead problem in detail and give an overview of the proposed method in Section 3. We propose a cosimulation method based on SW analysis in Section 4. We present experimental results and conclusion in Section 5 and 6, respectively.

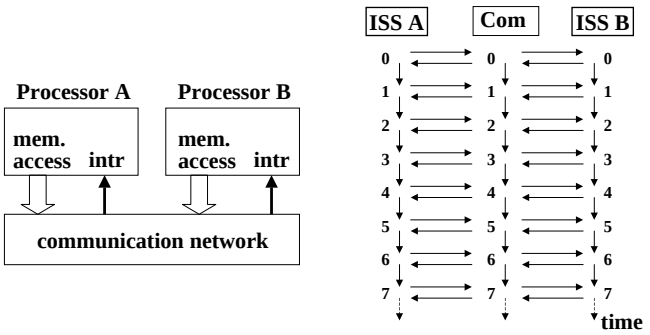
2 Related Work

As a performance improvement method of cycle-accurate cosimulation, Hines and Borriello present a concept called **selective focus** [5][6][7]. It enables the designer to trade off between performance and accuracy by changing the abstraction levels of communication models during cosimulation run. Liu et. al. present a method that re-uses cycle-accurate simulation results (delay information) of previous simulation runs [8]. For each run of cycle-accurate simulation of SW program running on a processor, its timing delay is stored in a **delay cache** according to the execution path of the simulated SW program. The stored delay information is re-used, when the same execution path is to be simulated, and in that case, functional simulation of the execution path is performed instead of cycle-accurate simulation. Like the selective focus concept, the method improves the simulation performance by sacrificing simulation accuracy.

To improve the performance of cycle-accurate cosimulation while preserving simulation accuracy, an effective concept called **memory image server** is used in commercial cosimulation environments [9]. The basic idea of the concept is that the ISS accesses the external memory model only when it accesses memory area mapped to HW-SW communication. For accesses to the other memory area, the ISS accesses its internal memory model. By doing that, the ISS can eliminate such external memory accesses as instruction fetching, accesses to local memory area, etc. However, the memory image server concept cannot reduce the synchronization overhead to detect the occurrence of interrupt. In commercial tools, a practical solution is requesting the designer to provide them with timing information such as the time interval of no interrupt occurrence. However, outside of the time interval, ISS's should synchronize with other simulators at every system clock or every instruction execution to detect the occurrence of interrupt.

To reduce synchronization overhead of interrupt detection, Yoo and Choi present optimistic simulation approaches [10][11][12]. They are effective in reducing the synchronization overhead, especially in geographically distributed cosimulation. However, one drawback of their approaches is a compatibility problem with commercial simulators since few commercial simulators support optimistic simulation features (i.e. state saving and restoration) [13][14].

In this paper, we present a method to reduce the synchronization overhead without sacrificing simulation accuracy. Moreover since the SW program analysis used in the method is based on conventional SW program analysis [15] and graph theory [16], for the



(a) An example of interupt-based inter-processor communication

(b) Lock-step synchronization

Figure 1: An example of synchronization overhead.

proposed method, we do not have to change the internal implementation of ISS's, but have only to change their memory access functions (in practice, the designer is allowed to modify memory access functions of ISS's to model their own memory models).

3 Problem and Solution Overview

3.1 Synchronization Overhead Problem

Figure 1 (a) shows an example of dual-processor target architecture. We assume that inter-processor communication (IPC) from one processor to the other is initiated by an access from one processor to memory area mapped to IPC (we call it **IPC memory area**). To notify the initiation of IPC, an interrupt is issued to the other processor through the communication network (e.g. point-to-point communication channel [17], shared bus communication channel with dedicated interrupt logic [3], etc). Figure 1 (b) shows an example of simulator synchronization. In cycle-accurate cosimulation, three simulators (two ISS's and one for the communication network) are assumed to be involved. Since interrupt is used for IPC, to detect the occurrence of interrupt, ISS's exchange messages (horizontal arrows in the figure) with the simulator of communication network (Com) at every system clock tick (vertical arrow in the figure). Since synchronization is performed in a lock step manner, we call such synchronization **lock-step synchronization**. Such multiple simulator synchronization causes frequent inter-process communications each of which is very time-consuming, thereby resulting in a significant degradation in cosimulation performance.

3.2 Solution Overview

One effective way of reducing the synchronization overhead is to reduce the number of messages exchanged between simulators for the synchronization. To do that, each ISS should know the time points when interrupts arrive at the processor that it is simulating. If it is possible, the ISS has only to synchronize with other simulators at such time points and at the time points when it makes external memory accesses. However, since interrupt is assumed to be triggered by a memory access of another processor, an ISS cannot tell exactly when such a memory access is executed by another processor. Thus, each ISS should notify other ISS's of its (predicted) time point of memory access for IPC. To do that, each simulator should be able to predict the time points when it makes memory accesses to IPC memory area.

In this paper, we present a prediction method based on instruction-level analysis of application SW. The method gives pessimistic prediction at every instruction, i.e. **the minimum execution time among those from the present instruction of application SW to all memory instructions accessing IPC memory area**.

For the prediction, we first calculate (1) minimum instruction execution time of each instruction and (2) minimum basic block execution times. Then we run Dijkstra's algorithm to compute the execution time from each instruction to the nearest memory instructions accessing IPC memory area. Those steps are done in each ISS before cosimulation starts. During cosimulation, each ISS notifies other simulators of the predicted execution times so that they can predict the time point of interrupt arrival. For multiple simulator synchronization, we modify an algorithm of conventional conservative distributed simulation [18].

4 Cosimulation based on SW analysis

In this section, we present the prediction method and the algorithm for multiple simulator synchronization.

4.1 Prediction based on SW analysis

4.1.1 Terminology and Assumptions

Before we describe the method, we define our terminology.

- IPC memory access instruction ($\mathrm{I^{IMA}}$): Assembly instruction that accesses IPC memory area.
- Minimum instruction execution time (MIET): The minimum time delay of an instruction execution.
- Basic block: A sequence of instructions with single entry point, single exit point, and no internal branch [15].
- B(I): The basic block to which instruction I belongs.
- Minimum basic block execution time (MBET): Sum of MIET's of all the instructions in the basic block.
- Basic block graph, BBG(BB, E, W): A directed graph with nodes (basic blocks) and edges (dependency between basic blocks).[1] BB is the set of basic blocks and E is the set of edges. The weight of each edge is the MBET of the source node of the edge. That is, $\mathrm{w_{i,j}} = \mathrm{MBET(n_i)}$ for $\mathrm{e_{i,j}}$, where $\mathrm{w_{i,j}}$ is the weight of the edge $\mathrm{e_{i,j}}$ between source node $\mathrm{n_i}$ and destination node $\mathrm{n_j}$ and MBET($\mathrm{n_i}$) is MBET of node $\mathrm{n_i}$. W is the set of weights.
- IMA node ($\mathrm{N^{IMA}}$): A node (i.e. basic block) that contains one or more $\mathrm{I^{IMA}}$'s.
- $\mathrm{T^{I2I}(I)}$: Minimum execution time from instruction I to the nearest $\mathrm{I^{IMA}}$.
- $\mathrm{T^{N2N}(I)}$: Minimum execution time from the entry point of B(I) to the entry point of basic block of the nearest $\mathrm{I^{IMA}}$.

We have the following two assumptions. Assumption 4.1 is already given in Section 3. We re-state it in this section for better reference.

Assumption 4.1 *Inter-processor communication (IPC) from one processor to the other is initiated by an access (mostly, write operation) to IPC memory area.*

Assumption 4.2 *The time delay from the start to the completion of IPC can be given in a time interval* $[\mathrm{ND_{min}}, \mathrm{ND_{max}}]$.

Assumption 4.2 can apply to most of multi-processor communication architectures such as point-to-point communication architectures [17], shared bus architectures with dedicated interrupt logic [3], and conventional multi-processor architectures [19]. For instance, in the case of shared bus architecture, $\mathrm{ND_{max}}$ can be the maximum delay of shared bus access considering bus contention and interrupt logic operation.

[1] We use two terms, node and basic block, interchangeably.

```
A:
0x0  mov  r2, #3
0x4  cmp  r2,r1
0x8  ble  D
B:
0xc  ldr  r0,[r13,#0x10]
0x10 ldr  r1,[r13,#0xc]
0x14 add  r0,r0,r1
0x18 cmp  r0,r1
0x1c blt  E
C:
0x20 b    C
D:
0x24 add  r2,r0,r1
0x28 str  r2,[r13,#4]
E:
0x2c mov  r0,#1
0x30 str  r0,[r13,#8]
0x34 b    C
```

(a) An example of assembly code

```
A:
0x0      1
0x4      1
0x8      3
B:
0xc      3
0x10     3
0x14     1
0x18     1
0x1c     3
C:
0x20     3
D:
0x24     1
0x28     2
E:
0x2c     1
0x30     2
0x34     3
```

(b) An example of MIET' s

Figure 2: An example of MIET calculation.

4.1.2 Intra-Basic Block Analysis

First, MIET of each instruction is calculated. In modern pipeline architectures of processor, MIET of single-cycle instruction is a single cycle. For multi-cycle instructions, we can consider two types of MIET estimation: static and dynamic. Static estimation of MIET can be applied to multi-cycle instructions such as multiple data transfer (multiple load, store, etc) instructions or instructions that have the program counter as an operand [20], where the number of execution cycles can be determined statically. For the instructions (e.g. multiply instruction using a booth multiplier) whose execution delay is determined dynamically (e.g. determined by the values of operands during the program execution), we set MIET of such an instruction to the minimum value (i.e. a single cycle).

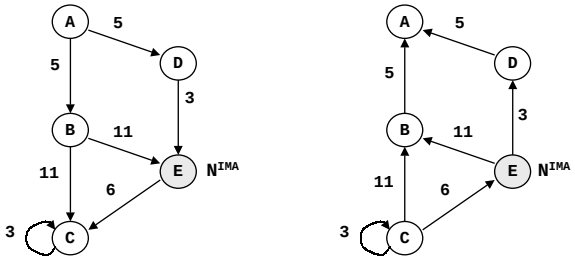
Figure 2 (a) shows an example of assembly code of ARM7 processor [20]. In Figure 2 (b), each instruction is given its MIET (a number on the right of each instruction). In the figure, each label corresponds to a basic block. MBET is the sum of MIET's in the basic block. For instance, basic block A has 5 as its MBET. In Figure 2 (a), we assume that the instruction located at `0x30`, "`str r0, [r13,#8]`" is an $\mathrm{I^{IMA}}$.

4.1.3 Inter-Basic Block Analysis

After intra-basic block analysis, we build a basic block graph, BBG (BB, E, W). Figure 3 (a) shows an example of basic block graph obtained from the code example in Figure 2 (a). Note that the weight on each edge corresponds to the MBET of the edge's source node. For instance, two edges going out of node A have 5 as their weights since the MBET of node A is 5. In the figure, node E is an $\mathrm{N^{IMA}}$ since it has an $\mathrm{I^{IMA}}$.

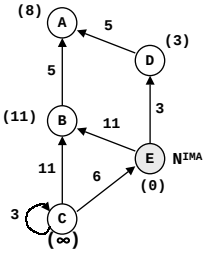
To calculate the minimum execution time ($\mathrm{T^{I2I}(I)}$) from any instruction I to the nearest $\mathrm{I^{IMA}}$, we first calculate the minimum execution time ($\mathrm{T^{N2N}(I)}$) from the entry point of the basic block (B(I)) of instruction I to all the entry points of IMA nodes. **Calculation of $\mathrm{T^{N2N}(I)}$ corresponds to the shortest path calculation in the BBG**. Since the number of destination nodes is usually much less than that of source nodes, it is easier to compute the shortest paths from the destination nodes to the source nodes. Therefore, we construct another graph called rBBG(BB, rE, W) by reversing the direction of each edge in the BBG. Figure 3 (b) shows an example of rBBG obtained from the BBG in Figure 3 (a). To calculate the shortest path from an $\mathrm{N^{IMA}}$ to every nodes (the entry points) in the rBBG, we apply Dijkstra's algorithm [16].

Figure 4 (a) shows the result of applying Dijkstra's algorithm to the rBBG in Figure 3 (b). In the figure, each node is given the length of shortest path, i.e. $\mathrm{T^{N2N}}$. Note that node C is given infinity (∞) since there is no execution path from node C to the $\mathrm{N^{IMA}}$ (node E).



(a) BBG(BB, E, W) **(b) rBBG(V, rE, W)**

Figure 3: Examples of BBG and rBBG.




```
A:
0x0      1, 8
0x4      1, 7
0x8      3, 4
B:
0xc      3, 9
0x10     3, 6
0x14     1, 5
0x18     1, 4
0x1c     3, 1
C:
0x20     3, ∞
D:
0x24     1, 3
0x28     2, 1
E:
0x2c     1, 1
0x30     2, 1
0x34     3, 1
```

(a) Result of Dijkstra algorithm **(b) An example of $\mathrm{T^{I2I}}$'s**

Figure 4: An example inter-basic block analysis result.

Then, for each instruction I, we calculate $\mathrm{T^{I2I}(I)}$ as follows.

$$\mathrm{T^{I2I}(I)} = \mathrm{T^{N2N}(I)} - \sum_{\mathrm{J \in B(I), J \geq I}} \mathrm{MIET(J)} + \sum_{\mathrm{J \in N^{IMA}, J > I^{IMA}}} \mathrm{MIET(J)}$$

where the relation J > I (J $\geq$ I) represents a control dependency between two instructions I and J, i.e. instruction J is executed earlier than (or at the same time with) instruction I. Figure 4 (b) shows the result of calculating $\mathrm{T^{I2I}(I)}$ for the example. In the figure, the third column corresponds to $\mathrm{T^{I2I}(I)}$'s. For instance, instruction at `0x4` has $\mathrm{T^{I2I}(I)}$ = 8 - 2 + 1 = 7.

Dijkstra's algorithm calculates the shortest path from a single source node to each node in a directed graph. In our case, since we have only to run Dijkstra's algorithm with each $\mathrm{N^{IMA}}$ as the source node, the complexity of applying Dijkstra's algorithm is $\mathrm{O(N^2) \cdot M}$, where N and M are the number of basic blocks and the number of $\mathrm{N^{IMA}}$'s, respectively (depending on the implementation of the Dijkstra's algorithm, the complexity can be slightly different). As shown in our experiments, the runtime overhead of static analysis is negligible.

4.2 Multiple simulator synchronization

A basic principle of multiple simulator synchronization is that the next pessimistic synchronization time point ($\mathrm{T^{sync}}$) is determined to be the lower bound of the time when the next $\mathrm{I^{IMA}}$ starts its execution. It is given by

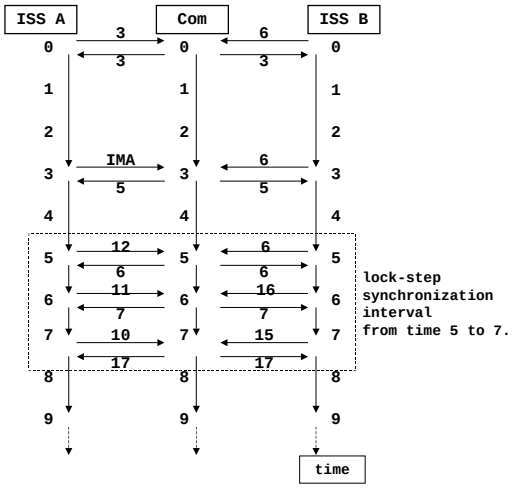


Figure 5: An example of multiple simulator synchronization.

$$T^{sync} = curr\_time + min_{\forall i \in Set\_of\_ISS's} T^{I2I}(curr\_inst(i))$$

where curr_time is the globally synchronized time and curr_inst(*i*) is the instruction to be executed currently in ISS *i*. In terms of implementing multiple simulator synchronization, there can be two ways of T^{sync} calculation: centralized or distributed calculation. In the centralized calculation, ISS's send their prediction values ($T^{I2I}(curr\_inst(i))$) to the central synchronization server (in practice, the simulator of communication network can be the central server) and the central server calculates T^{sync} and distributes it to each ISS. In the distributed calculation, each ISS calculates T^{sync} after receiving all the prediction values from the other ISS's. In this paper, we explain synchronization with the centralized calculation.

Figure 5 shows an example of multiple simulator synchronization where ISS A and B are involved. We assume that the communication between the two processors takes between 2 (minimum delay) and 4 (maximum delay) time units, i.e. ND_{min} = 2 and ND_{max} = 4. We also assume that, at time 0, ISS A sends $T^{I2I}(curr\_inst(A))$ = 3 (already calculated during the static analysis) and ISS B sends T^{I2I} (curr_inst(B)) = 6 to the simulator of communication network (**Com**) which acts as the synchronization server. **Com** determines time 3 as T^{sync} and sends the value to the two ISS's. They proceed simulation without synchronization until time 3.

At time 3, ISS A executes an I^{IMA} and sends to **Com** the information that it starts executing an I^{IMA}. Since ISS A starts execution of I^{IMA} at time 3 and the minimum network delay (ND_{min}) is 2, ISS B can receive an interrupt at the earliest at time 5 (= 3+2). Since ISS B still has 6, its previous prediction value, **Com** determines time 5 as T^{sync}. Therefore, simulators do not synchronize with each other until time 5.

Note that, during the time interval [5, 7], lock-step synchronization should be performed between **Com** and each ISS since the communication network is assumed to have minimum 2 time unit and maximum 4 time unit delay of interrupt trigger operation and it can trigger an interrupt to a processor during the time interval [3+2, 3+4]. We call such an interval **lock-step synchronization interval**. Figure 5 shows a lock-step synchronization interval (dashed rectangle). Note that each I^{IMA} execution is assigned a lock-step synchronization interval. At the end of the time interval - at time 7 in our example - ISS A sends $T^{I2I}(curr\_inst(A)) = 10$ and ISS B sends $T^{I2I}(curr\_inst(B)) = 15$ to **Com**. In this case, **Com** determines time 17 (=7+10 < 7+15) as T^{sync}. Until T^{sync}, simulators proceed without synchronization. Simulation continues in this way.

```
1 time_advance( ) {
2    if Tsync == time  {
3       receive TI2I or IMA info. from each ISS;
4
5       if there is any IIMA to be executed {
6          Tlock-step_min [i]= time + NDmin
7          Tlock-step_max [i++]= time + NDmax;
8       }
9
10      if any j, such that Tlock-step_min [j]<= time
11                          & Tlock-step_max [j]>= time
12         Tsync ++;   // during a lock-step synch. interval.
13      else
14         Tsync = min{(time + TI2I)'s, Tlock-step_min[j](> time)};
15
16      send Tsync to each ISS;
17   }
18   time++;
19 }
```

Figure 6: Pseudo code of centralized calculation of next synchronization point.

Table 1: Statistics of system examples.

	ISS 1		ISS 2	
	No. BB's	Code size	No. BB's	Code size
Config1	923	16 KB	443	11 KB
Config2	125	5 KB	170	5 KB
JPEG	426	10 KB	44	0.46 KB

In Figure 6, a pseudo code of centralized calculation of T^{sync}. In the figure, $T^{lock-step}_{min}[j]$ ($T^{lock-step}_{max}[j]$) is the start (finish) time point of lock-step synchronization interval related with the j-th I^{IMA} execution. From line 5 to 8, if there is a new I^{IMA} execution, a new lock-step synchronization interval is assigned to the I^{IMA} execution. At any time point, whether it is in a lock-step synchronization interval or not can be determined by the conditionals in line 10 and 11. Note also that T^{sync} is determined by the minimum of (time + T^{I2I})'s and $T^{lock-step}_{min}[j]$'s (which are later than the current time) as shown in line 14. Function **time_advance()** is called at every time tick in the synchronization server.

5 Experiments and Discussion

We apply the cosimulation method to two embedded systems: an IS-95 CDMA cellular phone system [21][22] and a JPEG encoder system [23]. As the ISS, we use a commercial ISS of ARM processor called ARMulator [24]. As the simulator of communication network (**Com**), we use our own cycle-based simulator. In our implementation, **Com** plays the role of central synchronization server.

The IS-95 system consists of four modules: voice encoder and decoder and CDMA modem transmitter and receiver. In our experiments, we performed cosimulation for the voice receiving path, i.e. CDMA modem receiver and voice decoder. We have three types of simulator configuration for the IS-95 system. In the first simulator configuration (Config1), we use two ISS's (one for a part of voice decoder and the other for the remaining part of voice decoder). In the second configuration (Config2), we simulate the CDMA modem receiver with two ISS's (one for a part of modem receiver and the other for the remaining part of modem receiver). In the third configuration (Config3), we run four ISS's (each ISS runs one of four parts used in Config1 and Config2). In the case of JPEG encoder system, we run two ISS's (one for discrete cosine transformation and the other for the remaining part of JPEG system).

Table 1 shows the statistics of SW programs running on each ISS. In the table, the case of Config3 is not shown since the four SW programs in Config1 and Config2 are used in the case. Table 2 shows the simulation results in the case of uni-processor cosimulation on a SUN UltraSPARC (167Mhz, 320MB). Table 3 shows

Table 2: Comparison of simulation results in the case of uni-processor cosimulation.

	Lock-step		Proposed method			Overhead	Reduction in	Runtime
	Runtime (sec)	No. msg's	Runtime (sec)	Overhead	No. msg's	ratio	no. msg's	speedup
Config1	405	3,439,518	81	0.7	558,596	0.9%	83.8%	5.0
Config2	692	5,495,650	45	0.2	108,770	0.5%	98.0%	15.4
Config3	1,509	12,613,600	180	0.2	1,259,480	0.1%	90.0%	8.4
JPEG	1,143	10,918,294	211	0.4	267,594	0.2%	97.6%	5.4

Table 3: Comparison of simulation results in the case of dual-processor cosimulation.

	Lock-step		Proposed method			Overhead	Reduction in	Runtime
	Runtime (sec)	No. msg's	Runtime (sec)	Overhead	No. msg's	ratio	no. msg's	speedup
Config1	2,786	3,439,518	457	0.4	558,596	0.09%	83.8%	6.1
Config2	4,390	5,495,650	111	0.1	108,770	0.1%	98.0%	39.6
Config3	3,932	12,613,600	403	0.2	1,259,480	0.05%	90.0%	9.8
JPEG	8,445	10,918,294	352	0.2	267,594	0.06%	97.6%	24.0

the simulation results in the case of dual-processor cosimulation on two workstations (SUN UltraSPARC, 167Mhz, 320MB and SUN UltraSPARC, 200Mhz, 1GB). In the experiments, we assumed that the communication delay is one cycle (i.e. equivalent to the delay of a fast point-to-point communication channel).

As the tables show, we obtained up to orders of magnitude higher performance in cosimulation runtimes by reducing significantly the number of inter-process messages (up to 98%) with negligible runtime overhead ($\leq$ 0.9%). Note that for Config1, the reduction of the number of messages is relatively small. It is because the vocoder consists of many small loops. In the proposed method, we use the shortest path algorithm to estimate the minimum execution time to the IMA node and assume that loops are executed only once for the estimation of the shortest execution time. Therefore, in Config1, the estimated minimum execution times can fall short of the real minimum execution times more frequently than in other configurations.

In the case that network delay is difficult to be given as an interval or another HW simulator(s) is involved in cosimulation, it will be hard to obtain improvement by applying our method directly. One way to get around is (1) to apply the proposed method to the synchronization in a cluster of simulators where ISS's and the communication network simulator satisfy the two assumptions (Assumption 4.1 and 4.2) and (2) for the designer to specify regions in the SW and HW where no interrupt occurs, which is the strategy taken by commercial tools such as Seamless CVE [9]

6 Conclusion

In this paper, we proposed a method to improve the cosimulation performance of multi-processor embedded systems. The method achieves the performance improvement by reducing simulator synchronization overhead. To do that, before cosimulation run, a static analysis of application software is applied to predict simulator synchronization time points. We applied the method to the cosimulation of practical embedded systems and obtained up to orders of magnitude higher performance in cosimulation runtimes.

References

[1] B. Clement, R. Hersemeule, E. Lantreibecq, P. Coulomb, B. Ramandin, and F. Pogodalla, "Fast Prototyping: a System Design Flow Applied to a Complex System-On-Chip Multiprocessor", *Proc. Design Automation Conf.*, pp. 420–424, 1999.

[2] T. W. Albrecht, J. Notbauer, and S. Rohringer, "HW/SW CoVerification Performance Estimation & Benchmark for a 24 Embedded RISC Core Design", *Proc. Design Automation Conf.*, pp. 808–811, June 1998.

[3] J-Y. Brunel, W.M. Kruijtzer, H.J.H.N. Kenter, F. Petrot, and L. Pasquier, "COSY Communication IP's", *Proc. Design Automation Conf.*, pp. 406–409, June 2000.

[4] W. Richard Stevens, *UNIX Network Programming*, Prentice Hall, 1991.

[5] K. Hines and G. Borriello, "Optimizing Communication in Embedded System Co-simulation", *Proc. Int. Workshop on Hardware-Software Codesign*, pp. 121–125, Mar. 1997.

[6] K. Hines and G. Borriello, "Selective Focus as a Means of Improving Geographically Distributed Embedded System Co-simulation", *Proc. Eighth IEEE International Workshop on Rapid System Prototyping*, pp. 58–62, June 1997.

[7] K. Hines and G. Borriello, "Dynamic Communication Models in Embedded System Co-Simulation", *Proc. Design Automation Conf.*, pp. 395–400, June 1997.

[8] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, "Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator", *Proc. Int. Workshop on Hardware-Software Codesign*, pp. 65–69, Mar. 1998.

[9] Mentor Graphics, Inc., "Seamless CVE", available at http://www.mentorg.com/seamless/ .

[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 and K. Choi, "Optimizing Timed Cosimulation by Hybrid Synchronization", *Design Automation for Embedded Systems*, vol. 5, no. 2, June 2000.

[13] Synopsys, Inc., *Cyclone VHDL Reference Manual*, Synopsys Online Documentation, v1998.08.

[14] Quickturn, "PowerSuite", *available at http://www.quickturn.com/products/psbroch.htm*.

[15] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[16] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, McGraw-Hill, Inc., New York, 1994.

[17] W. Wade, "Embedded chips diverge on multiprocessing path", *EE Times, Issue 1118*, June 2000.

[18] R. Bagrodia, K. M. Chandy, and W. T. Liao, "A Unifying Framework for Distributed Simulation", *ACM Trans. on Modeling and Computer Simulation*, vol. 1, no. 4, pp. 348–385, Oct. 1991.

[19] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., New York, 1993.

[20] D. Jaggar, *Advanced RISC Machines Architectural Reference Manual*, Prentice Hall, July 1996.

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

[22] 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*, pp. 61–66, Oct. 1999.

[23] Portable Video Research Group, "PVRG-JPEG CODEC", *available at ftp://havefun.stanford.edu/pub/jpeg/JPEGv1.2.1.tar.Z.*

[24] ARM Ltd., "Software Development Toolkit", *available at http://www.arm.com /products/SDT/.*