Trace-Driven HW/SW Cosimulation Using Virtual Synchronization Technique

Dohyung Kim
dhkim@ucsd.edu

Youngmin Yi
ymyi@iris.snu.ac.kr

Soonhoi Ha
sha@iris.snu.ac.kr

School of Computer Science and Engineering, Seoul Nation University, 151-742, Seoul, Korea

ABSTRACT

Poor performance of HW/SW cosimulation is mainly caused by synchronization requirement between component simulators. Virtual synchronization technique was proposed to remove the need of synchronization in cycle accurate cosimulation. But the previous execution-driven simulation based on virtual synchronization has limitations in the application area. In this paper, we propose a novel trace-driven HW/SW cosimulation using virtual synchronization technique. Through OS modeling and channel modeling, the proposed cosimulation technique could be applied more widely while improving the simulation performance further. Experiments with a DIVX player example prove the viability of the proposed technique.

Categories and Subject Descriptors

B.7.2 [**Design Aids**]: Simulation and Verification

General Terms: Performance, Verification

Keywords: Trace-driven cosimulation, virtual synchronization,

1. INTRODUCTION

Hardware/software (HW/SW) cosimulation verifies the system performance replacing real processing components with component simulators running concurrently. Since faster estimation of the system performance promises wider design space exploration, boosting the simulation speed has been a major focus in HW/SW codesign research. It is also the main theme of this paper.

Performance of HW/SW cosimulation depends on the speed of component simulators and synchronization overhead between them. Figure 1 shows the experimental result. We have applied three different synchronization methods that have huge difference in synchronization overhead. Three curves in Figure 1 display the simulation time of component simulators in the total elapsed time of cosimulation as we vary the simulator speed for three different synchronization methods. When the simulator speed is 18M cycles per second and synchronization is achieved by a function call, 10% of the total elapsed time is due to the simulation time of the component simulators and 90% is spent for synchronization. Therefore the synchronization overhead becomes the major bottleneck of cosimulation as the simulator performance increases.

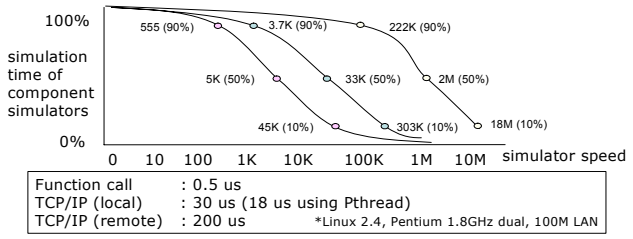

Figure 1. (Simulation time of the slowest simulator/total cosimulation time) versus the speed of the slowest simulator adopting different synchronization methods

We have proposed the virtual synchronization technique [1][2] to reduce the synchronization overhead almost to zero by removing the synchronization requirement between component simulators. The basic idea of the virtual synchronization technique is to separate global time management of simulation from local simulators utilizing algorithm model. When a local simulator produces output samples, the time differences between output samples are recorded from the simulator. The actual global time of output samples is computed by the simulation backplane.

However the technique has the following limitations. Since it is an execution-driven simulation, the simulation backplane should wait to receive output data from all component simulators to safely advance the global time considering the resource contention between components. Moreover static delays are assumed for local and shared memory accesses and no blocking is allowed during the execution of a task.

The main contribution of this paper is *to combine the trace-driven simulation and virtual synchronization techniques* while the previous works used execution-driven simulation. It consists of two parts running concurrently. In the first part, it captures the execution traces from processing components ignoring the global time management. In the second part, it reconstructs the global time information for cycle accurate simulation behavior using trace-driven cosimulation. The proposed trace-driven simulator models both the communication architecture and the OS behavior. Blocking behavior can also be considered in trace-driven simulation. As a result, we can maintain good performance of virtual synchronization while overcoming the limitations of the previous execution-driven simulation in terms of application domain and modeling accuracy. Another contribution of this paper is to propose a formulation on the speed of a conventional conservative cosimulation technique as discussed in section 2. The proposed formula gives insight to understand the diverse cosimulation techniques to improve the performance.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

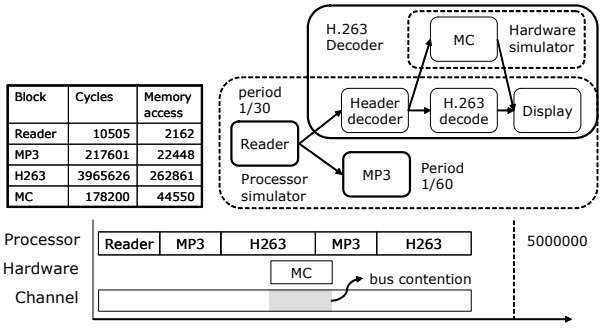

Figure 2. Motivational example: DIVX player

2. ANALYSIS OF COSIMULATION TIME

In this chapter, we extract key factors that affect cosimulation performance and present a simple formula on cosimulation time. We separate the simulation of processing components and the simulation of communication architecture. This separation makes performance analysis of cosimulation environment easier because two simulations have different performance factors. And different accuracy levels can be applied to enhance cosimulation performance

In HW/SW cosimulation, the simplest approach to achieve accurate timing behavior is to synchronize all component simulators at every clock. It is called a conservative approach and most existent cosimulators use this approach for its simplicity. So we formulate the cosimulation time of a conservative cosimulation approach as definition 1. In equation (1), simulation time to advance one clock cycle, synchronization overhead, transaction count, and transaction simulation time are recognized as the major performance factors. Simulation time of a simulator depends on the design complexity as well as the modeling accuracy of the simulator.

Definition 1. Analysis of conservative cosimulation time

T : Total simulated cycles

st_i : Simulation time to advance one cycle of simulator i

$sync$: Synchronization overhead

$tran_{num}$: The total number of communication transactions

st_{tran} : Simulation time to process a transaction

$$\sum_{\forall i}\{T \times st_i + T \times sync)\} + tran_{num} \times st_{tran} \quad \textbf{(1)}$$

Figure 2 shows a motivational example, DIVX player, which is composed of three tasks: reader task, H.263 decoder task and MP3 decoder task. The reader task is invoked at every 1/30 second and sends data packets to other tasks. The H.263 decoder task is activated when data is available and the MP3 decoder task invoked at every 1/60 second. In the H.263 decoder task, the motion compensation (MC) block is mapped to the hardware simulator (HS) and all other tasks are executed on the processor simulator (PS). At the left table of Figure 2, information on the execution cycle and the memory access count of each task is shown. At the bottom of Figure 2, a runtime schedule is displayed.

In equation (2), the cosimulation time of conservative simulation of the DIVX player is computed based on equation (1).

$$T = 5M,\ st_{PS} = 1\,us,\ st_{HS} = 100us,\ sync = 40us, tran_{num} = 332021,\ st_{tran} = 154\,us$$
$$5M \times \{(1\,us + 40us) + (100us + 40us)\} + 332021 \times 154\,us = 956s \quad \textbf{(2)}$$

Although performance parameters are 10 times faster than measured from Seamless CVE [3], the cosimulation speed is less than 6Kcycles/s (5M/956s).

3. RELATED WORK

In this section, we review the related approaches from the viewpoint of how they improve the performance factors in equation (1). Transaction level model (TLM) enhances simulation performance adapting higher abstraction level in communication and computation respectively, which reduces synchronization points and simulation times (st_i, st_{tran}) [4]. However, cycle-accurate cosimulation of TLM approach performs still less than 100Kcycles/s because of conservative synchronization and operating system running on the processor simulator.

Trace-driven simulation [5] stores memory traces from cycle-accurate cosimulation without considering any delay due to resource contention. Then, it evaluates different communication architectures very fast (almost zero st_i and synchronization overhead) by simulating dynamic behavior from communication architecture using memory traces. However, trace-driven simulation requires large external storages which become the performance bottleneck of the approach. In addition, it can not consider dynamic behavior from multiple processors and operating systems because of pre-determined trace order.

In optimized synchronization [6], all simulators notify the next event time to one another. Then each simulator can safely advance its time until the smallest next event time of simulators. In optimistic synchronization [7], each simulator advances its local clock assuming that no event will arrive. If this assumption fails, it rolls back its local time to the event arrival time canceling all the results that have been processed after that time. These approaches reduce synchronization overhead (T x $sync$) but their application area and performance enhancement are limited by their prerequisites.

SeamlessCVE [3] reduces synchronization overhead (T x $sync$) by making only shared memory accesses delivered to the communication simulator (hardware simulator). But it loses time accuracy without considering bus contention for local memory accesses and still shows poor performance.

4. TRACE-DRIVEN COSIMULATION

The proposed trace-driven HW/SW cosimulation consists of two parts (Figure 3). In the first part, the simulation engine executes tasks on component simulators, captures execution traces, and provides them to the task representatives of the second part of the cosimulator. The trace-driven simulator models the behavior of operating system and communication architecture. If the trace-driven simulator consumes all input traces or cannot determine the next trace to evaluate, it requests new execution traces to the simulation engine of the first part and waits until they arrive.

4.1 Trace Generation

Capturing execution traces is done by executing tasks on component simulators (software or hardware) according to the algorithm model as shown in Figure 2. An execution of a task continues until it is blocked by data or by period. During task execution the memory model of each simulator stores memory traces and important events of task behavior as behavior traces. After each task execution, the component simulator delivers execution (memory, behavior) traces to the simulation engine as shown in Figure 3. Since a component

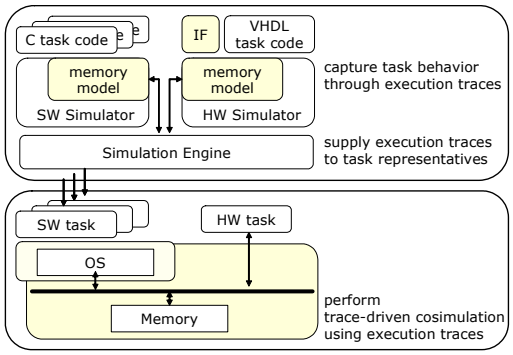

Figure 3. Framework for trace-driven cosimulation

simulator usually gives a way to define the memory model for global memory access, no modification on the component simulator itself is not needed.

Note that the execution traces contain the local time information of the component simulators. In the first part global time need not be managed. So each trace stores the time difference compared with the previous trace. It also means that *each simulator does not need to advance its local clock when they have nothing to execute*. An example of trace generation is illustrated in Figure 4 where absolute times of traces are meaningless.

For SW simulator, assignments to the special control address are automatically annotated to generate behavior traces in the memory model. In HW simulator, the memory model catches control signals from interface codes (IF) to generate behavior traces. Table I shows trace types and their descriptions of six behavior traces and four memory traces that are currently recorded in the proposed cosimulator.

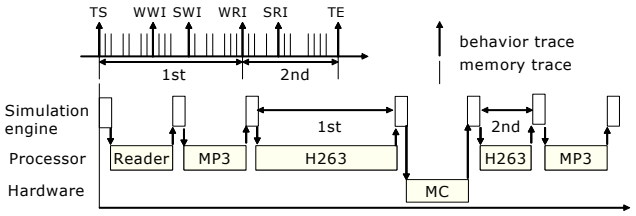

Figure 4. Capturing execution traces

Table I. Trace types and their description

Behavior trace type	Behavior trace description
TASK_START (TS)	indicate the start of a task
TASK_END (TE)	indicate the end of a task
WAIT_READ_INTR (WRI)	wait data availability
WAIT_WRITE_INTR (WWI)	wait buffer availability
SEND_READ_INTR (SRI)	notify buffer availability
SEND_WRITE_INTR (SWI)	notify data availability
Memory trace type	Memory trace description
SEQ_READ (SR)	sequential read
NSEQ_READ (NSR)	non-sequential read
SEQ_WRITE (SW)	sequential write
NSEQ_WRITE (NSW)	non-sequential write

4.2 Trace-Driven Cosimulation

Using execution traces obtained from the simulation engine, we perform trace-driven cosimulation. Compared to the trace-driven simulation approach [5], our approach requires only a small portion

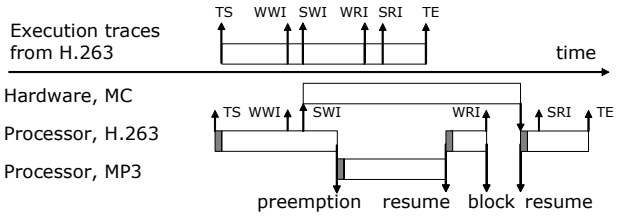

Figure 5. Operating system model scenario

of execution traces because trace generation and consumption are performed simultaneously. So there is no storage and performance problem. Moreover the proposed trace-driven cosimulation models dynamic behavior using behavior traces such as preemptions by other tasks and interrupts from other processors.

Trace-driven cosimulation is further divided into two steps. First, the operating system model determines a sequential order of execution traces from all tasks for each processor. It also computes operating system delays like interrupt overhead and context switch overhead, and applies them to the sequence. Because every occurrence of behavior traces in any processor (interrupts from other components, blocked on channel, and end of task) affects the sequence of execution traces, the OS model selects the next candidate task with the earliest active time of each processor every time when it happens.

Figure 5 shows an example scenario of OS model. OS model starts the H.263 decoder task (TS) when data is available from the reader task. The MP3 decoder task with a higher priority preempts the H.263 decoder task at the next tick interrupt. The H.263 decoder task is resumed when the MP3 task ends. Also note that the H.263 decoder task is blocked (WRI) until data become available from the MC block. Finally OS model ends the execution of the H.263 decoder task.

Second the channel model resolves the conflicts on the communication architecture. It chooses a processor with the earliest access time among processors. Then it computes inter-connection and memory latency for memory traces. It delivers an interrupt to the destination processor for SEND_WRITE_INTR or SEND_READ_INTR behavior traces. Once the communication architecture produces an additional delay by bus contention and memory, all memory traces after the point are postponed by the delay if the access times are overlapped as shown in Figure 6.

The OS model and the channel model can be easily modified to model different architectures. Our previous research [2] shows that the OS model without cache is achieved very accurately under 0.1% error and with cache at moderate error level (under 7%). The channel model in the trace-driven cosimulation is similar to that in transaction based channel model [4] that is successfully achieved through transaction level modeling.

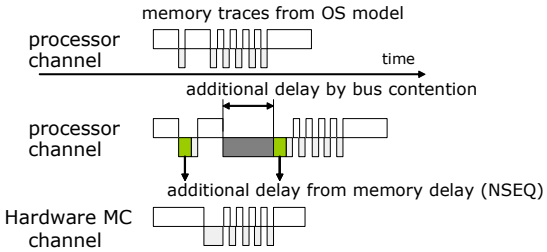

Figure 6. Channel model example

4.3 Performance Analysis

Definition 2 explains how virtual synchronization achieves huge performance enhancement compared to other approaches. First, because each simulator is not synchronized with other simulators during trace generation, it greatly reduces synchronization counts of the simulators (sc_i). Second, because an execution trace stores a time difference to the previous trace, the local clock of each simulator does not need to be synchronized with other simulators. It means that a simulator need not execute idle duration of the processor ($u_i \times T$). Lastly, efficient implementation of trace-driven simulation makes the communication simulation very fast by having small st_{tran}.

Definition 2. Cosimulation time of the proposed approach

u_i : Utilization of simulator i

sc_i : Synchronization counts of simulation i

$$\sum_{\forall i} \{ T \times u_i \times st_i + sc_i \times sync \} + tran_{num} \times st_{tran} \quad \textbf{(3)}$$

Compared to cosimulation time of the conservative approach in equation 2, trace-driven cosimulation achieves 42% performance gain without time synchronization and 51% without idle duration as calculated in equation 4. As a result, it shows 43 times better performance.

$$T = 5M,\ st_{PS} = 1\ us,\ u_{PS} = 88\%,\ st_{HS} = 100us,\ u_{HS} = 3.6\%$$
$$sc_{PS} = 5,\ sc_{HS} = 1,\ sync = 40us, tran_{num} = 332021,\ st_{tran} = 0.75\ us$$
$$(5M \times 1\ us \times 0.88 + 5 \times 40us) + (5M \times 100us \times 0.036 + 1 \times 40us)\} +$$
$$332021 \times 0.75\ us = 22.4s \quad \textbf{(4)}$$

5. EXPERIMENTS

We use ARMulator for processor simulators and ModelSim for a hardware simulator on Linux 2.4 with Xeon 2.6Ghz dual CPUs. Table II shows different architectures for the DIVX player and Table III cosimulation results of decoding 20 frames for each architecture. In Table III, the second column shows simulation times, the third column shows simulated cycles and the final column shows simulator performances as cycles per second.

Table II. Different architectures from the partition step

	FPGA	Arm922T (Proc 1)	Arm922T (Proc 2)
Arch. 1			All tasks
Arch. 2		IDCT	Other tasks
Arch. 3		MC	Other tasks
Arch. 4	IDCT		Other tasks
Arch. 5	IDCT	MC, MP3	Other tasks

Table III. Cosimulation performance: simulation time, simulated cycles w/o considering utilization, performance

	Time	Cycles	Speed
Arch. 1	64s	100,076,232	1,564K
Arch. 2	109s	119,053,960	1,092K
Arch. 3	79s	89,678,834	1,135K
Arch. 4	168s	101,553,023	604K
Arch. 5	173s	72,601,086	420K

Table IV. Performance comparison with other approaches from Arch. 4 for decoding 3 frames

	App.	Cycles	Time	Speed
Virtual sync.	DVIX	14944028	26s	575K
Seamless CVE	H.263	8839060	9755s	0.91K
CoCentric Studio	M-JPEG	10185864	303s	33K

Table IV shows performance comparison for the architecture 4 with commercial cosimulation tools. Although the result comes from a little different application and environment, it could be regarded reasonable because the architecture has similar design complexity. The result shows that our approach is even much faster than a TLM simulator, CoCentric Studio from Synopsys.

6. CONCLUSION

The proposed approach based on virtual synchronization technique separates generation of execution traces and timing management of execution traces in HW/SW cosimulation. It removes synchronization overhead between component simulators and boosts simulator speed about two orders of magnitude compared with the conservative approach. Implementation in PeaCE provides automatic generation of cosimulation environments for different architectures and design steps.

Reduced synchronization overhead enables effective execution of distributed simulators at the simulator engine. So the advantage of the proposed technique will be greater for multi-processor SoC targets. Since the simulation accuracy depends on the OS model and the channel model, future research will be focused on the modeling of real systems.

7. ACKNOWLEDGMENTS

This work was supported by National Research Laboratory Program (Grant No. M1-0104-00-0015) and IT R&D Project funded by Korean MIC. The ICT and ISRC at Seoul National University provided research facilities for this study.

8. REFERENCES

[1] D. Kim, C. Rhee, and S. Ha, "Combined Data-driven and Event-driven Scheduling Technique for Fast Distributed Cosimulation", IEEE Transactions on VLSI Systems Vol. 10 pp 672-679 Oct. 2002

[2] Y. Yi, D. Kim, S. Ha, "Fast and Time-Accurate Cosimulation with OS Scheduler Modeling", Design Automation for Embedded Systems, Kluwer Academic Publishers Vol. 8 pp 211-228 Sep. 2003

[3] B. Bailey, "Co-Verification: From Tool to Methodology," Mentor Consulting Technical Publication, June 2002

[4] T. Grotker, S. Liao, G. Martin and S. Swan, "System Design with SystemC", Kluwer Academic, Norwell, Mass., 2002

[5] K. Hines and G. Borriello, "Optimizing communication in embedded system cosimulation", in Proc. Intl. Symp. on Hardware/Software Codesign, pp.121-125, Mar. 1997.

[6] Wonyong Sung and Soonhoi Ha, "Optimized Timed Hardware Software Cosimulation without Roll-back", DATE 98, Paris, France February 1998

[7] S. Yoo and K. Choi, "Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model", Proc. of Proc. 6th Int'l Workshop on HW/SW Co-Design, Mar. 1998