Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model

Sungjoo Yoo Kiyoung Choi

School of Electrical Engineering
Seoul National University
Seoul 151-742, Korea
{ysj,kchoi}@poppy.snu.ac.kr

Abstract

In this paper, we present thread-based optimistic distributed timed cosimulation methods which reduce the overhead of optimistic simulation. First, we present thread simulation model to facilitate efficient distributed cosimulation. To reduce the overhead of optimistic simultion, we focus on the reduction of state saving overhead. Based on the thread simulation model, we perform thread-level state saving without saving the whole state of processor at each checkpoint. Especially, single checkpoint property based on the proposed thread model minimizes the number of state savings for HW threads. We give preliminary experimental results to show the efficiency of the proposed methods.

1 Introduction

In the current design practice of HW-SW systems, cosimulation is a representative method for covalidation. As the complexity of HW-SW systems grows, the covalidation of given system specification gets more difficult and time-consuming, and becomes the bottleneck of achieving shorter time-to-market. Distributed cosimulation is an attractive approach to improve cosimulation performance through performing parallel simulation on a distributed simulation environment such as a multi-processor workstation and a network of workstations.

In timed cosimulation, synchronization overhead caused by the frequent exchange of local times between simulators becomes dominant in cosimulation run-time [1]. In distributed timed cosimulation, the synchronization overhead gets bigger due to high overhead communication between simulators on different processors or workstations. Moreover, as the simulation performance of simulator itself gets better by performing simulation at higher abstraction levels or using a simulation accelerator [2] [3], the relative portion of synchronization overhead in total cosimulation run-time gets much bigger. In such a case, conservative distributed timed cosimulation may suffer from high synchronization overhead to give less simulation speedup than expected [4].

Optimistic simulation has advantage in such a case that synchronization overhead is dominant [5]. However, in optimistic simulation, maximum simulation performance is determined by the overhead of optimistic simulation such as state saving overhead. In this paper, we present efficient state saving methods based on thread model to reduce the overhead of optimistic distributed timed cosimulation.

This paper is organized as follows. In section 2, we give a review on distributed cosimulation. In section 3, we give a thread simulation model. In section 4, we propose methods to reduce the overhead of optimistic distributed timed cosimulation. We show the efficiency of our methods in a case study in section 5. We give conclusion and future work in section 6.

2 Related Work

To improve simulation performance, distributed simulation concept has been applied to cosimulation. Thomas [6] exploits the parallelism of a pipelined system performing dual process cosimulation. Ghosh [7] presents a cosimulation environment in which distributed cosimulation can be performed in a lock step manner. Hines [8] uses *selective focus* to improve the performance of geographically distributed cosimulation. Valderrama [9] presents a distributed cosimulation environment based on VCI, an automatic cosimulation interface generation tool.

There have been few research work on applying optimistic simulation concept to timed cosimulation. In [1], optimistic simulation is performed to minimize the

1092-6100/98 $10.00 © 1998 IEEE

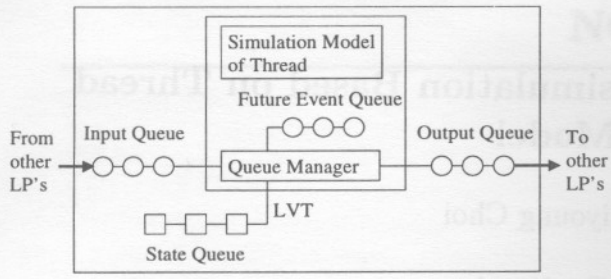

Figure 1: Logical Process.

number of simulator synchronization in single processor timed cosimulation. [8] allows optimistic simulation to be performed locally in a component simulation and synchronization between components is performed in a synchronous manner.

3 Thread as A Simulation Model

In our work, we consider a thread as a code segment of instructions in SW (SW thread) and a set of physical components in HW (HW thread) satisfying the following assumptions.

Assumption 3.1 Threads communicate by *initiation/interrupt/resumption signals* and *read/write operations.*

The execution of an *instance* of a thread is initiated, interrupted, and resumed by initiation signal, interrupt signal, and resumption signal coming from other threads. Interrupt signal is sent to only SW threads. If a SW thread receives an interrupt signal, its execution is interrupted and an *interrupt service thread* starts to execute. An interrupted thread resumes its execution when it receives a resumption signal.

Assumption 3.2 A thread reads data from *input buffer(s)* only when the execution of its instance starts. It writes results to *output buffer(s)* only just before the execution of the instance ends.

Assumption 3.2 holds in practical systems such as embedded real-time systems in which tasks starts their execution reading data from input buffers and finishes the execution writing results to output buffers and done signal to a status register [10]. For SW threads, however, Assumption 3.2 can be too restrictive in such cases as semaphores are used to serialize the access to critical sections. We apply Assumption 3.2 only to HW threads and allow SW threads to perform read/write operations at any point of their execution.

In optimistic simulation, a set of *logical processes* (LP's) execute concurrently and communicate by exchanging timestamped events, or *messages.* A message is represented by a tuple <`msg_id, msg_type, time, event`>, where `msg_id` enables each message to be distinguished, `msg_type` can be *positive message* or *negative (or anti-) message*, and `time` is the time when `event` will be evaluated. As a component of message, `event` represents a signal or read/write operation described in Assumption 3.1. For each thread, we define *state* as the simulation image of the thread.

We assign a logical process to each thread. An LP contains several objects as shown in Figure 1. In Figure 1, Local Virtual Time (LVT) is the time associated with the LP. Future Event Queue (FEQ) is an event queue used when there are internal events scheduled within the LP itself. Input Queue (IQ) is a message queue which has messages sent to the LP by other LP's. Output Queue (OQ) is a message queue which has messages sent to other LP's. State Queue (SQ) contains states stored for the case of rollback.

IQ contains incoming messages which have timestamps earlier than LVT as well as messages having timestamps later than LVT. Messages having timestamps earlier than LVT are kept in IQ for the case of rollback of the LP. OQ also contains messages sent to other LP's for the case of rollback of the LP. The queue manager and the simulation model of thread work as follows.

1. The queue manager looks up messages in IQ.
   (a) If it finds a *straggler message* (a message having timestamp earlier than LVT) in IQ, then it restores from SQ the latest state having timestamp earlier than or equal to the timestamp of the straggler message and sets LVT to the time of the restored state.
   (b) If we take aggressive cancellation to output messages, then the queue manager sends a canceling message per each message in OQ having timestamp earlier than that of the straggler message.
2. The simulation model of thread executes events scheduled at the present LVT including the events in messages in IQ. LVT is updated after executing event(s).
3. (a) Send output messages (if any) to other LP's. If we take lazy cancellation to output message, then new output messages are compared to old ones to be canceled. For old output messages which do not match new ones and have timestamps equal to LVT, anti-messages are sent to the corresponding LP's.
   (b) If LVT is a checkpoint, then the current state of the LP is stored in SQ. Goto Step 1.

An anti-message to a message having a signal as its event represents that since the previous signal was not correct, the initiation/interrupt/resumption of an instance should be canceled. For the case of read/write operation, an anti-message represents that since the previous data read/written or the time when the data is read/written was not correct, read/write operation should be performed again.

Global virtual time (GVT) is defined as the minimum of timestamps of *in-transit* messages[1] and local virtual times of all LP's. Messages and states having timestamp earlier than GVT are removed since a logical process will never be rolled back to a timestamp earlier than GVT [11]. However, only one state having the latest timestamp earlier than or equal to GVT is kept for the case of rollback.

4 Reduction of State Saving Overhead

In a physical viewpoint, the state of a thread represents the content stored in memory and registers of processors (SW or HW processors) of the simulated system while the thread is executing. In embedded systems such as multimedia systems the size of memory accessed by SW and HW processors can be several megabytes. In such a case, state saving overhead can be dominant in optimistic distributed cosimulation. To reduce the overhead of state saving, the amount of content to be stored should be as small as possible.

Thread model enables state saving to be performed on a thread basis, which makes it possible that only the states of currently running threads need to be practically stored at each checkpoint. Moreover, utilizing the non-interruptible nature of HW threads we can minimize the number of state saving in case of HW threads. Note that our methods assume Copy State Saving which stores states by copying the simulation image after the execution of each event or periodically.

4.1 State Saving of HW Threads

Property 4.1 *For a HW thread, we have only to perform single checkpoint at the end of execution of each instance.*

During the execution of i-th instance of HW thread $I(i)$, a straggler message arriving at the LP which simulates the instance $I(i)$ represents one of two cases:

- Case 1 : The previous initiation signal to $I(i)$ should be canceled.
- Case 2 : The data read by $I(i)$ was not correct.

[1]Messages which are in the communication channels between LP's, or not processed yet in input queues.

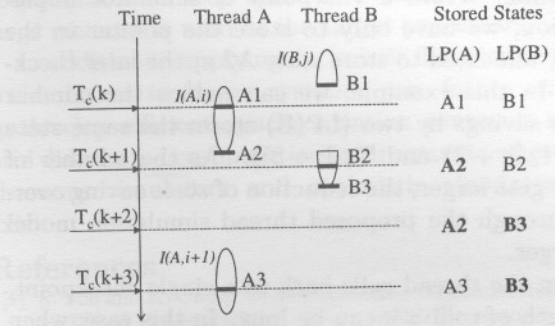

Figure 2: State Saving of SW Threads.

For Case 1, the LP must cancel the execution of $I(i)$. For Case 2, the LP must re-execute $I(i)$ by reading the correct data. For each case, the LP cancels all the execution since the initiation of $I(i)$, and rolls back to the initiation point of the instance execution due to Assumption 3.2 for HW threads. Since the LP rolls back only to the initiation point of instance execution, we have only to keep the state of the thread at the end of execution of the previous instance $I(i-1)$ to support the rollback.

4.2 State Saving of SW Threads

Since SW threads can read/write at any point of their execution, single checkpoint property can not be applied to them. Figure 2 shows how states of SW threads are stored on a thread basis. We assume that two threads, thread A and B are running on the same SW processor and the execution of the j-th instance of thread B, $I(B,j)$ is interrupted by the i-th instance of thread A, $I(A,i)$. Checkpoints are set at time points $\mathrm{T_c(k)}$, $\mathrm{T_c(k+1)}$, $\mathrm{T_c(k+2)}$, and $\mathrm{T_c(k+3)}$. LP(A) and LP(B) represent logical processes which simulate thread A and B, respectively.

At checkpoint $\mathrm{T_c(k)}$, thread A is running and thread B is interrupted. LP(A) stores state A1, the state of running thread A. LP(B) stores state B1, the state of interrupted thread B. At checkpoint $\mathrm{T_c(k+1)}$, since the execution of instance $I(A,i)$ ended before the checkpoint, LP(A) stores state A2 which was stored at the end of execution of instance $I(A,i)$. LP(B) stores state B2 of running thread B. At checkpoint $\mathrm{T_c(k+2)}$, since both of two threads finished the execution of their instances before $\mathrm{T_c(k+2)}$, LP(A) and LP(B) store the states A2 and B3, respectively.

In this example, since LP(A) stores the same state A2 at $\mathrm{T_c(k+1)}$ and $\mathrm{T_c(k+2)}$, we do not have to save state A2 at the later checkpoint $\mathrm{T_c(k+2)}$. Instead, we have only to keep the same state A2 at the two

checkpoints. From a viewpoint of simulator implementation, we have only to store the pointer to the memory allocated to store state A2 at the later checkpoint. In this example, we can reduce the number of state savings by two (LP(B) stores the same state B3 at $T_c(k+2)$ and $T_c(k+3)$). As the number of threads gets larger, the reduction of state saving overhead through the proposed thread simulation model gets larger.

When the thread rolls back to a single checkpoint, the length of rollback can be long. In this case, when a new message comes in after rollback occurs, we can *jump forward* to the timestamp of the new message.

5 Experiment

In our experiment, we use a compiled model of TMS320C50 DSP processor [1] for SW simulation, and a cycle-based simulator[2] for HW simulation. We developed a distributed simulation library and compiled HW and SW simulator with the library. We use **UNIX socket** for the communication between simulators.

We performed two types of timed cosimulation: uni-processor synchronous timed cosimulation and optimistic distributed timed cosimulation. In optimistic distributed timed cosimulation, we used two SUN Sparcstation-4's (32 Mbyte main memory) on 10 Mbps Ethernet LAN. The SW simulator and the motor simulator are allocated on one workstation and the HW simulator is on the other. We assume that the SW processor has 4 Kbyte memory.

5.1 Example : A CNC machine

As an example system, we use a real-time servo control system, a CNC (Computer Numerical Control) machine [12]. The system controls a mechanical device (cutter) which moves on a flat two-dimensional plane. Motions on the X and Y axes are carried out by two separate servo motors. The system consists of 7 tasks. In our experiment, one task is implemented in HW, the others are in SW. Since tasks in the CNC machine satisfy the assumptions in Section 3, we treat each task as a thread.

For the timed cosimulation of the CNC machine, we use a CNC motor simulator [12] together with the SW simulator and the HW simulator. Between the motor simulator and the SW simulator, the location value of the cutter (from motor to SW) and the command for motor control (from SW to motor) are transferred, which is repeated at the rate of 30 μs in reality. We set the system clock of the controller to 10 MHz. We perform timed cosimulation of one rotation of the plant which takes 0.6 second in reality.

[2]Developed at Seoul National Univ.

Table 1: Speedup Comparison between Two Types of Timed Cosimulation (in sec).

	synchronous	optimistic
SW	135	135 + 31 (OV)
Motor	50	50
IPC	4551	107 (SW), 80 (HW)
HW	211	211 + 34 (OV)
Total	5012	325

Since the motor simulator is a synchronous simulator which does not perform optimistic simulation, the SW simulator sends the command to the motor when the global virtual time of HW and SW simulators becomes greater than or equal to the periodic synchronization points set by the motor simulator.

5.2 Comparison of Simulation Run-time

Table 1 shows the run-time of two types of timed cosimulation. Comparing the computation load of uniprocessor synchronous cosimulation (135+50+211 = 396 seconds without IPC overhead) and the simulation run-time of distributed cosimulation (325 seconds), we obtained 1.22 times speedup through optimistic distributed cosimulation. In terms of the total simulation run-time, optimistic distributed timed cosimulation gives more than 15 times speedup.

In Table 1, OV represents the overhead of optimistic simulation in HW and SW simulation. The overhead includes managing input/output queues, state saving, rollback, and re-execution. Since we assume the SW processor has a quite small memory, 4K byte memory, state saving overhead is very low (less than 1%) in this experiment. We will show how the size of SW processor memory affects the total simulation run-time in the next subsection.

In uni-processor synchronous timed cosimulation, interprocess communication (IPC) overhead dominates in simulation run-time. It is because each of HW and SW simulators sends/receives synchronization information to/from the other simulator on every clock tick. Although the communication overhead between processes on the same workstation is much smaller than the overhead over an Ethernet LAN, the number of synchronization is very high (in this example, 12,000,000) in synchronous timed cosimulation [1].

In optimistic distributed timed cosimulation, the number of simulator synchronization reduces dramatically as shown in Table 2. Among the types of message, the number of messages to request GVT calculation dominates. It is because the optimistic SW simulator should synchronize with the synchronous motor simulator at every 30 μs of simulated time.

Table 2: Message Statistics in Optimistic Distributed Timed Cosimulation.

no. total messages	no. positive messages	no. anti-messages	no. GVT messages
53,507	4,425	180	49,875

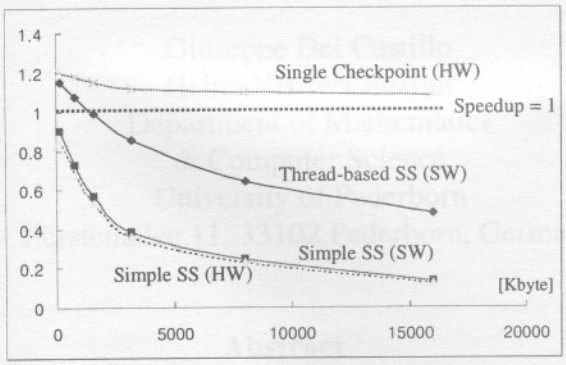

Figure 3: Speedup v.s. Size of Memory.

5.3 State Saving Overhead

To show the overhead of state saving of the proposed method as the size of memory used in the target system increases, we performed experiments varying (1) the size of memory used in the HW thread (task) and (2) the size of memory of SW processor.

Figure 3 shows the change of speedup of optimistic distributed timed cosimulation as the size of SW processor memory and the state size of the HW thread increases. We assume that the size of memory used in each SW thread increases at the same rate of the increase of SW processor memory.

In Figure 3, simple state saving means state saving which makes a copy of the whole memory of SW processor or the state of the HW thread at each checkpoint. As the size of memory increases, speedup drops rapidly in the simple state saving. In the thread-based state saving for SW threads, speedup decreases much slower than the simple state saving. For the HW thread, speedup keeps almost constant even in the case of large memory. Since our example is not a big one, the speedup is relatively low. We think that we can obtain sufficient speedup in bigger systems, especially, in which computation load is large.

6 Conclusion

In this paper, we present thread-based optimistic distributed timed cosimulation methods to reduce the overhead of optimistic simulation. Our preliminary implementation gives drastic speedup by performing distributed cosimulation on two workstations. Thread-based state saving methods enable distributed cosimulation to keep speedup even in the case that large memory is used in the target processors.

In addition to applying optimistic distributed timed cosimulation to larger systems, our future work is to develop efficient synchronization protocols in hybrid distributed cosimulation environments where synchronous simulators and optimistic simulators coexist.

References

[1] 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.

[2] K. Hines, "Pia: A Framework For Embedded System Cosimulation with Dynamic Communication Support", Technial Report, University of Washington, Oct. 1996.

[3] C. J. DeVane, "Efficient Circuit Partitioing to Extend Cycle Simulation Beyond Synchronous Circuits", *Proc. Int. Conf. on Computer Aided Design*, pp. 154–161, Nov. 1997.

[4] W. Sung and S. Ha, "Optimized Timed Hardware Software Cosimulation without Roll-back", *to appear in Proc. Design Automation and Test in Europe*, Feb. 1998.

[5] H. Rajaei, R. Ayani, and L. Thorelli, "The Local Time Warp Approach To Parallel Simulation", *Proc. 7th Workshop on Parallel and Distributed Simulation*, pp. 119–126, 1993.

[6] D. E. Thomas and S. L. Coumeri, "A Simulation Environment for Hardware-Software Codesign", *Proc. Int. Conference on Computer Design*, pp. 58–63, Oct. 1995.

[7] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamoto, "A Hardware-Software Co-simulator for Embedded System Design and Debugging", *Proc. Asia South Pacific Design Automation Conference*, 1995.

[8] 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.

[9] C. A. Valderrama, P. Lemarrec, and A. A. Jerraya, "VCI: A VHDL-C Interface Generation Tool for Cosimulation", *Proc. IEEE International High Level Design Validation and Test Workshop*, pp. 142–148, Nov. 1997.

[10] V. J. Mooney and G. De Micheli, "Real Time Analysis and Priority Scheduler Generation for Hardware-Software Systems with a Synthesized Run-Time System", *Proc. Int. Conf. on Computer Aided Design*, pp. 605–612, Nov. 1997.

[11] D. R. Jefferson, "Virtual Time", *ACM Tans. on Programming Languages and System*, vol. 7, no. 3, pp. 404–425, 1985.

[12] N. Kim, M. Ryu, S. Hong, M. Saksena, C. Choi, and H. Shin, "Visual assessment of a real-time system design: a case study on a CNC controller", *Proc. IEEE Real-Time Systems Symposium*, pp. 300–310, Dec. 1996.