Fast and Accurate Cosimulation of MPSoC Using Trace-Driven Virtual Synchronization

Youngmin Yi, *Member, IEEE*, Dohyung Kim, and Soonhoi Ha, *Member, IEEE*

***Abstract*—As MPSoC has become an effective solution to ever-increasing design complexity of modern embedded systems, fast and accurate cosimulation of such systems is becoming a tough challenge. Cosimulation performance is in inverse proportion to the number of processor simulators in conventional cosimulation frameworks with lock-step synchronization schemes. To overcome this problem, we propose a novel time synchronization technique called trace-driven virtual synchronization. Having separate phases of event generation and event alignment in the cosimulation, time synchronization overhead is reduced to almost zero, boosting cosimulation speed while accuracy is almost preserved. In addition, this technique enables (1) a fast mixed level cosimulation where different abstraction level simulators are easily integrated communicating with traces and (2) a distributed parallel cosimulation where each simulator can run at its full speed without synchronizing with other simulator too frequently. We compared the performance and the accuracy with MaxSim, a well-known commercial SystemC simulation framework, and the proposed framework showed 11 times faster performance for H.263 decoder example, while the error was below 5%.**

***Index Terms*—HW/SW cosimulation, multiprocessor system-on-chip (MPSoC), parallel simulation, SystemC, system simulation, virtual synchronization.**

I. Introduction

HARDWARE/SOFTWARE cosimulation is the key enabler of successful hardware/software codesign methodology in embedded system design [1]. HW/SW cosimulation validates both functional and timing correctness of the system before it is actually prototyped, replacing real processing components with component simulators that interact with one another. Since faster validation of the system performance promises wider design space exploration, boosting the cosimulation speed has been a major focus in HW/SW codesign research.

As MPSoC (Multiprocessor System-on-Chip) has become an effective solution to ever-increasing design complexity of modern embedded systems, fast and accurate cosimulation of such complex systems is becoming a tough challenge. With a conventional lock-step synchronization scheme, cosimulation performance is in inverse proportion to the number of the processor simulators. The Fig. 1 depicts the overall cosimulation performance measured in a commercial SystemC cosimulation framework, MaxSim [3]. We executed a matrix multiplication example using up to 8 ARM processor simulators on a 1.8 GHz Dual-Xeon simulation host. All the simulators are executed serially on one simulation host: if the simulated cycles are T, each simulator executes its T cycles serially on the same simulation host so that the total simulation time is the sum of simulation times of all component simulators and context switching overhead.

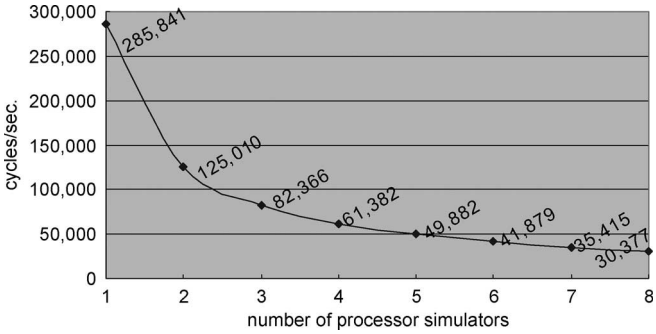


Fig. 1. Overall SystemC cosimulation performance is in inverse proportion to the number of processor simulators.

In this paper, we present a fast cosimulation framework that employs ISSs for MPSoC systems. There are mainly three approaches to achieve fast cosimulation: (1) increasing the speed of a simulator itself; (2) reducing time synchronization overhead among the simulators, and (3) executing the simulators in parallel on distributed simulation hosts. We propose a novel time synchronization technique, trace-driven virtual synchronization to replace lock-step synchronization. With this scheme, time synchronization overhead is reduced almost to zero while accuracy is preserved within a certain bound. The basic idea of the technique is to decouple event generation and event alignment in the cosimulation. Each simulator executes its own task(s) with as little synchronization as possible and the cosimulation backplane aligns the generated events, reconstructing the global time. Trace-driven virtual synchronization technique correctly simulates complex and dynamic behavior of system by repeating the event generation phase and event aligning phase in the cosimulation.

Performance improvement by adopting faster simulators is complementary to the one of reducing time synchronization overhead. The proposed cosimulation framework further improves cosimulation performance by employing fast simulators such as SystemC simulation models. On the other hand, not

Manuscript received November 21, 2006; revised February 15, 2007. This work was supported in part by the BK21 project, by the SystemIC 2010, by the IT-SoC, and by the KOSEF research program (R17-2007-086-01001-0). This paper was recommended by Associate Editor L. Benini.

Y. Yi is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720 USA (e-mail: ymyi@eecs.berkeley.edu).

D. Kim is with Google, Inc., Seoul 135-090, Korea (e-mail: dohyung@google.com).

S. Ha is with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-600, Korea (e-mail: sha@iris.snu.ac.kr).

Digital Object Identifier 10.1109/TCAD.2007.907048

only for faster speed but also for design reuse, it is necessary to adopt different abstraction level simulators. If IPs are given in a fixed level, mixed level cosimulation is inevitable. The trace-driven feature of the proposed time synchronization scheme enables easy integration of different abstraction level simulators. In addition, the reduced number of time synchronization enables fast mixed level cosimulation.

Finally, the proposed cosimulation framework achieves the higher performance through parallel cosimulation. Distributed parallel cosimulation requires efficient time synchronization between multiple simulator processes on different simulation hosts or on different cores in a multicore machine. With trace-driven virtual synchronization, simulators can be executed in parallel increasing the overall performance.

The rest of this paper is organized as follows. Section II reviews related work. Section III defines terminology used in this paper and formulates the conventional cosimulation performance with lock-step time synchronization. Section IV explains the proposed trace-driven virtual synchronization technique. Section V presents mixed level cosimulation and parallel cosimulation technique based on the proposed scheme. In Section VI, the experimental results are discussed. Finally, we conclude this paper in Section VII.

II. Related Work

Research on HW/SW cosimulation has been actively conducted during the past decade. We categorize and review previous studies in time synchronization perspective: early heterogeneous simulation, homogeneous simulation including SystemC simulation, recent heterogeneous cosimulation for mixed level cosimulation, and distributed parallel simulation. Then, we finally review trace-driven simulation.

Early HW/SW cosimulation environments were heterogeneous simulation environments where the HDL simulator and processor simulator communicate with each other using interprocess communication (IPC) [4], [5]. In connecting heterogeneous simulators, the cosimulation backplane approach is presented in [6] as opposed to pairwise-direct coupling. It plays the role of master process to manage the IPC between software simulator and hardware simulator and to integrate a new simulator seamlessly.

Lock-step simulation of heterogeneous cosimulation environment degrades the cosimulation performance significantly since it involves huge IPC overheads for time synchronization. To reduce per-synchronization cost of heterogeneous cosimulation, several approaches were suggested that use the same language for both software and hardware. The timing-annotated C code is used as a model for both hardware and software components in [7]–[9]. It is compiled as a host binary and run directly on a host workstation. Similarly, [10] presents VHDL-based HW/SW cosimulation where all the system is included in an HDL simulator; software is also modeled in VHDL (behavioral level) with delay information and hardware is modeled in RTL.

In recent years, transaction level modeling (TLM) [11], [12] has received a lot of attention. Reference [13] presents C/C++-based design environment for HW/SW coverification using SystemC. SystemC simulation is a homogeneous cosimulation environment where both software and hardware models are modeled in a single language, built as a single process in the host workstation. SystemC simulation is faster than traditional heterogeneous cosimulation since not only the abstraction level has been raised to transaction level from RTL but also time synchronization overhead between simulation models has been reduced from IPC overhead to thread switch overhead. Since TLM can provide various abstraction levels, high-level cosimulation using delay annotated host code execution in SystemC is presented in [14].

However, for more accurate cosimulation, ISSs are typically used in SystemC cosimulation [2], [3], [15], [16]–[19], which is also the main target of focus in this paper. ISSs are attached to a cycle accurate transaction level communication architecture model and communicate via IPC. Thus, reducing the number of synchronization has become important again. In [19], time synchronization at every N cycle has been suggested instead of every single cycle in a lock-step time synchronization scheme. However, this sacrifices cosimulation accuracy with the reduced time resolution.

In addition to the adoption of ISSs in SystemC framework, several studies present more general mixed-level cosimulation also including an RTL hardware simulator in SystemC framework. In [20] and [21], ISSs and RTL hardware simulators are connected to SystemC environment via IPC, and different abstraction level simulators are cosimulated. Since SystemC plays the role of backplane, time synchronization is done in a lock-step manner; each external simulator exchanges events with the SystemC interface module at every system cycle and this binds the cosimulation speed to the slowest simulator speed.

We now review time synchronization studies of parallel discrete event simulation (PDES). There has been a lot of research on this theme and they can be classified largely into two approaches [22]: conservative [23] and optimistic [24]. These approaches reduce the number of IPC for time synchronization and make distributed parallel cosimulation feasible.

The conservative approach guarantees that no past event will occur by advancing the local clock of a simulator in such a way that it cannot be larger than the minimum timestamp in the event queues of the incoming links. Thus, the simulators can run in parallel at their full speed without any synchronization until that timestamp value. If an event queue of any incoming link is empty, the simulator has to block and this can lead to deadlock. One solution to resolve the deadlock is that each simulator has to exchange a null message, which contains only a timestamp without any event. A null message is a promise of sending simulator that it will not send an event whose timestamp is smaller than the one in the null message. That is, null message overhead is time synchronization overhead and reducing the number of null messages is directly related to the performance of parallel discrete event-driven simulation. If the null message is exchanged at every cycle, this is lock-step simulation. To reduce the null message overhead in a conservative approach, it is necessary to increase the timestamp of the null message by predicting the next ordinary message.

An optimized conservative approach [25] was proposed to estimate the system-wide next earliest event. In this approach, a

simulator may advance its local clock up to the estimated next earliest event time without worrying about the occurrence of a past event. However, it is not always possible to estimate the future event time.

In [26], the time synchronization point is predicted based on a static analysis of application software running on each processor. It identifies in the application the interprocessor communication instructions and statically predicts minimum execution cycles from the current instruction to those IPC instructions. The predicted minimum timestamp, or lookahead in other words, should be large in order to get better improvement. Reference [27] employs a similar approach as in [26] but it extends lookahead using dynamic execution path prediction and hardware prediction. Branch prediction and the loop iteration count prediction template and lookahead table are obtained at compile-time and evaluated at run-time. Run-time prediction extends lookahead to several tens or hundreds of clock cycles of the target processor.

Reference [28] presents parallel cosimulation of multiple processor simulators with conservative time bucket synchronization scheme [22]. In this method, target execution is broken up into fixed lock-step intervals called quanta. It is assumed that target messages sent during one quantum can only affect target state in the subsequent quanta.

Optimistic approach, on the other hand, allows each simulator to advance its local clock optimistically assuming that no past event will arrive. If this assumption fails, it rolls back its local time to the latest checkpoint time canceling all results that have been processed after that time [29]. This approach is costly to maintain the internal states at each check-point time. In addition, if a processor simulator does not support a roll-back mechanism, as usually is the case, it cannot be used.

In [30], the system time is maintained as the minimum value of the local time of each component simulator. Time synchronization is performed optimistically only at communication through the input port and only when the system time and the local time are the same. However, if an interrupt is triggered and its time stamp is earlier than the local time, this scheme performs roll-back. A similar concept called memory image server is used in a commercial cosimulation environment [31]. This technique performs time synchronization only when simulators access a predefined memory region. However, it cannot correctly handle the occurrence of interrupt since it does not employ a roll-back mechanism.

The time synchronization idea presented in this paper is similar to one in [26], [27], and [30] in that it performs time synchronization only at interprocessor communication (intertask communication, to be exact). However, the proposed scheme is clearly distinguished from the previous works in that local time is not synchronized to global time and the local clock is only used as a timer to measure the difference between events. Decoupling of local time and global time enables the correct handling of the interrupts without roll-back, while still performing time synchronization only at intertask communication (The detailed explanation will be given in Section IV-B). In addition, the previous approaches would require synchronization at every local memory access as well as shared memory access in order to simulate the correct timing of communication architecture such as bus arbitration delay when contention occurs. The proposed approach can consider such behavior without synchronizing at every memory access but only at intertask communication, thereby maintaining high cosimulation speed.

Trace-driven simulation consists of trace collection and trace processing and these steps are separated and performed without any feedback in most cases [32]. Even if they may interact with the others, they aim to reduce the slowdown of the simulation and not for the correct and accurate simulation of dynamic behavior of the system. Such inaccuracy of trace-driven simulation of the multiprocessor is pointed out in [33]. Reference [34] addresses trace-driven cosimulation for rapid design space exploration of mapping of application models onto architecture models. Application is modeled in Kahn Process Network and trace events are coarse grain and generated only from three functions in YAPI [35] (read, write, and execute). It is similar to traditional trace-driven simulation in that trace generation and trace evaluation do not interact.

Trace-driven virtual synchronization is clearly distinguished from the traditional trace-driven simulation approach in that trace generation and trace evaluation parts interact for time synchronization, or global time alignment of traces. The proposed approach can reflect complex and dynamic behavior including OS scheduling policy and contention in the bus, since such architectural influence in the trace evaluation phase is repeatedly fed back to the trace generation phase.

We have proposed an execution-driven virtual synchronization technique [36] based on the following assumptions: (1) the execution result depends only on the arrival order of input events, not on the absolute arrival times; (2) the intertask or interprocessor communication occurs at the boundary of task execution and must be a nonblocking operation; (3) communication architecture has static delay. As a result, it was only applicable to dataflowlike applications that consist of tasks with nonblocking operation while assuming simple and static communication architecture. In this paper, we overcome these limitations by making the virtual synchronization technique trace-driven not execution-driven.

III. Terminology

In this section, we formally define the terms that we will use to describe the proposed synchronization scheme in later sections. In addition, we formulate the cosimulation performance with lock-step synchronization scheme as reference to compare it later with the proposed one. In the definition of terms, we borrowed the notations from [37].

A. Terminology

$w_i(x)\nu$	Write of a value ν at address x by simulator i.
$r_i(x)\nu$	Read of a value ν at address x by simulator i.
$\mathrm{GC}(e)$	Timestamp of event e in global clock.
$\mathrm{Execute}(e, e')$	Event e is executed before event e'.

Definition 3.1. Causal Order: We define that an event, e, is in causal order to another event, $e'(e \rightarrow e')$ if one of the following conditions holds:

1) $\exists i, j((e = w_i(x)\nu) \wedge (r_j(x)\nu = e') \wedge (\mathrm{GC}(e) < \mathrm{GC}(e')))$
2) $\exists e''(e \rightarrow e'' \wedge e'' \rightarrow e')$

The first condition of definition 3.1 describes true dependency between events and the second one describes the transitive relation of the order.

Definition 3.2. Causality Constraint: If an event, e, is in causal order to another event, e', then a simulator or simulators must process e before e'. This can be simply denoted as follows:

$$\forall e, e' \, (e \rightarrow e' \Rightarrow \text{Execute}(e, e')) \, .$$

Definition 3.3. Out-of-Order Simulation: It is defined as follows:

$$\exists e, e' \, ((\text{GC}(e) < \text{GC}(e')) \wedge (\text{Execute}(e', e))) \, .$$

B. Analysis of Cosimulation Time With Lock-Step Scheme

In the lock-step synchronization scheme, simulators synchronize at every cycle to satisfy causality constraint. It can be denoted as follows:

$$\forall e, e' \, (\text{GC}(e) < \text{GC}(e') \Rightarrow \text{Execute}(e, e')) \, .$$

We formulate the cosimulation time of a lock-step time synchronization approach as definition 3.4 [40].

Definition 3.4. Cosimulation Time With Lock-Step Scheme:

N	Number of simulators.
T	Total simulated cycles.
st_i	Simulation time to advance one cycle of simulator i.
sync	Overhead per time synchronization.
T_{trans}	Total number of communication transactions.
st_{trans}	Simulation time to process a transaction.

$$\sum_{\forall i}^{N} \{T \times (\text{st}_i + \text{sync})\} + T_{\text{trans}} \times \text{st}_{\text{trans}}. \qquad (1)$$

In (1), the simulation time to advance one clock cycle in a simulator, the overhead per time synchronization, the number of transactions, and the simulation time for a transaction are recognized as the major performance factors. We distinguish the overhead per time synchronization (sync) and the simulation time to process a transaction for data synchronization $(\text{st}_{\text{trans}})$.

IV. Trace-Driven Virtual Synchronization

A. Proposed Idea and Implementation

Definition 3.1 says that if an event, e, is in causal order to another event, e', then the global time of e is smaller than that of e'. However, if the global time of an event is smaller than the other, they are not necessarily in causal order: That is, $\text{GC}(e) < \text{GC}(e') \Rightarrow e \rightarrow e'$ is not necessarily true. For such events e, e', we can perform out-of-order simulation. The previous approaches that fall into conservative approach exploit this to reduce time synchronization overheads but the main obstacle is the asynchronous occurrence of interrupts. The proposed trace-driven virtual synchronization technique reduces time synchronization overhead as much as possible even when interrupts are used, by synchronizing only between events in causal order. For this goal, the proposed scheme decouples event generation and event alignment.

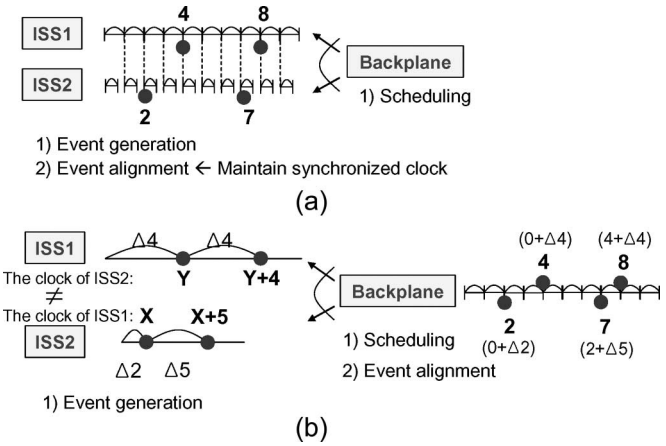


Fig. 2. (a) In previous approach, simulators both generate and align events. (b) In the proposed approach, simulators only generate events and the backplane aligns the events.

The key difference of the proposed approach and the previous ones is depicted in Fig. 2. In previous approaches, simulators both generate events and also align them. To correctly align the events and to prevent the occurrence of the past event, each clock of simulators must be synchronized [Fig. 2(a)]. On the contrary, in the proposed approach, simulators only generate events and the cosimulation backplane aligns the events releasing the simulators from the burden of the clock synchronization [Fig. 2(b)].

In the event generation phase, each simulator generates events such as local memory access without synchronizing with the cosimulation backplane until it encounters intertask communication such as shared memory access or completion of its execution. Since simulators only play the role of event generation and events are aligned globally in the backplane, each simulator does not need to synchronize its clock to the global clock. Instead, timestamp of an event is set as the relative difference of execution time (i.e., simulated cycles) from the previous event occurrence. Hence, the clock of a simulator is rather considered to be a timer than a clock and used only to measure the difference. Event information including timestamp is kept as a form of trace and delivered to the cosimulation backplane at the synchronization point.

In the event alignment phase, the cosimulation backplane reconstructs the global time of the generated events, or traces, by adding the time difference provided in a trace to the clock of the component that the trace belongs to. Each clock of components is maintained inside the backplane. Then, it aligns the traces conservatively up to the maximum possible global time; it first find out the earliest event (i.e., the trace with the minimum timestamp) among all the components. Timestamp of events from different components can now be compared since it has been translated to global time. Trace-driven simulation is performed with the earliest event and the global clock is advanced during the trace-driven simulation. The backplane continues the global time reconstruction and the alignment through trace-driven simulation until any event queue gets empty. To guarantee incremental event alignment and to prevent

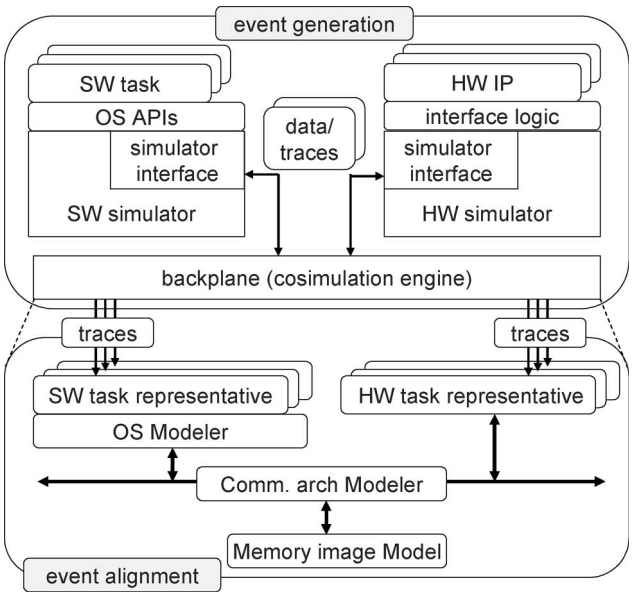


Fig. 3. Cosimulation framework for trace-driven virtual synchronization.

past event occurrence, the backplane must choose the earliest event. If any event queue is empty, the event alignment is paused and the event generation phase is invoked again. The backplane schedules each simulator in such a way to obey definition 3.2. By repeating these two phases, global time is advanced conservatively up to the given end time after all.

Fig. 3 illustrates the structure of cosimulation framework with trace-driven virtual synchronization. Each simulator has its own simulator interface that connects to the cosimulation backplane and generates traces in the event generation phase. It reads input data from the backplane and sends output data and traces to the associated representative in the backplane. A representative is maintained per task and simply keeps the traces. The total number of the tasks in the target system and the mapping information is statically provided to the backplane along with the relevant parameters such as a task priority, if necessary. The backplane creates task representatives based on the provided information when the cosimulation starts.

Trace-driven simulation is performed in the backplane and it is further divided into two steps: OS modeling and communication architecture modeling. Both models are provided by the backplane from the library and can be parameterized by the designers.

OS modeling [38] is required since, in our framework, a simulator executes its application tasks nonpreemptively without synchronization. To correctly reflect the occurrence of interrupts, the OS modeler determines a sequential order of events of all tasks in each processor by emulating the preemption behavior caused by the preemptive RTOS scheduler or interrupts. It also reflects RTOS delays such as interrupt handling overhead and context switch overhead that are given as parameters. In our current implementation, OS modeler can model priority-based preemptive scheduling and round-robin time-slice scheduling. Note that, in Fig. 3, the representative that keeps the traces is maintained per task rather than per component for the OS modeling purpose.

A communication architecture modeler [39] computes interconnection and memory latency for memory traces, taking into account the conflicts on the communication architecture. The memory access trace can be viewed as a transaction request to the communication architecture and the modeler provides cycle count accuracy at transaction boundary by adding the delay of communication architecture components. This kind of modeling is similar to the ones in [40] and [41]. However, the modeler in this paper does not resort to any discrete event simulation kernel but the modeler is defined as a function in the backplane. The input parameters to the modeler include the topology and the list of the components, the attributes of the components, and the address maps of communication architecture.

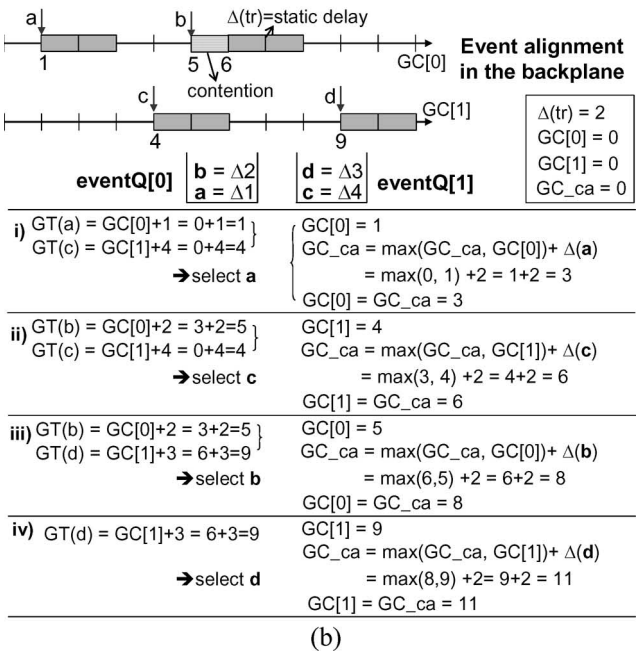
When a trace is given, the modeler first takes the address in a trace and finds out the component it is trying to access referencing the address maps. Then, it figures out the path from the requesting component to the destination component referencing the topology information. Along the path, it adds the time consumed on each communication component considering the bus arbitration policy and the contention. Currently, the modeler supports only AHB buses.

The simplified version of the trace aligning algorithm with the OS modeler and the communication architecture modeler is shown in Fig. 4. The algorithm is applied to every trace until either cosimulation finishes (line 1) or there is no trace left in a task representative (line 7). First, it finds out a task that accesses the communication architecture at the earliest time, translating the timestamp of each event into the global time (line 9–11). Global time reconstruction is performed in a function named GT() by adding the relative time in a trace to the global time of the component (line 16–18). It examines all the components in the system (line 3), considering OS scheduling in the components (line 4). After the earliest event has been identified, it is passed to the communication architecture modeler and the global time is advanced during the modeling (line 14, 19–24).

The Fig. 4(b) shows an example of the event alignment. Suppose that there are two components in the system and the traces "a" and "b" are generated from the first component and are queued in eventQ[0], while traces "c" and "d" from the other are queued in eventQ[1]. The clock of each component in the backplane and the clock of the communication architecture are denoted as GC[0], GC[1], GC}_ca and are all set to 0 initially. As explained, the backplane first finds out the earliest event that requests a transaction to the communication architecture. It translates the event time to the global time and compares the time of the events from GC[0] and GC[1]. Since GT(a) is earlier than GT(c), trace "a" is selected and is passed to the communication architecture modeler. First, it advances the clock of the component that the trace belongs to by the request time of the trace (GC[0] becomes 1). Then, trace-driven simulation is performed. In this example, for brevity, the static delay $(= \Delta(\mathrm{tr}))$ of the communication is uniformly assumed to be 2. To model the dynamic delay (i.e., arbitration delay) caused by the contention, the clock is set to the larger value among the current time of the communication architecture and the current component time (i.e., request time). Finally, the component time is advanced up to the communication architecture time.

```
1   while (cosim_end==false) {  // for each trace
2       // for each components
3       for (i=start_idx; i<num_components; i++) {
4           task = OS_Modeler( i );
5           if (task->trace == NULL) {    //eventQ is empty
6               start_idx = i;
7               return;                 // activate trace generation part
8           }
9           if (GT(task->trace) < min_access_time)
10              min_access_time = GT(task->trace);
11              min_task = task;   // find the earliest event
12          }
13      }
14      CommArch_Modeler( min_task->trace );
15  }
16  unsigned GT(trace *tr) { //reconstruct to the global time
17      return GC[tr->proc] + tr->time;
18  }
19  CommArch_Modeler(trace *tr) {
20      //advance the global clock
21   ... GC[tr->proc] = GT(tr);
22      GC_ca = max(GC_ca, GC[tr->proc]) +Δ(tr);
23      GC[tr->proc] = GC_ca;    ....
24  }
```

(a)



(b)

Fig. 4. (a) Trace alignment algorithm with OS modeler and communication architecture modeler and (b) Example.

In such a way, the traces are correctly aligned in the backplane in "a," "c," "b," and "d" order.

Note that if the request time of the current trace [e.g., trace "b" in Fig. 4(b)] is earlier than the completion time of the previous trace [e.g., trace "c" in Fig. 4(b)], the modeler postpones the access of the current trace and this extra delay accounts for the bus arbitration delay due to the contention.

The Fig. 5 shows a more complex example scenario of the proposed cosimulation and how the dynamic behavior of the system such as task blocking, preemption, and bus contention can be correctly simulated with significantly reduced synchronization overhead. Fig. 5(a) shows both the real behavior of the system and the cosimulation result. Assume there are two

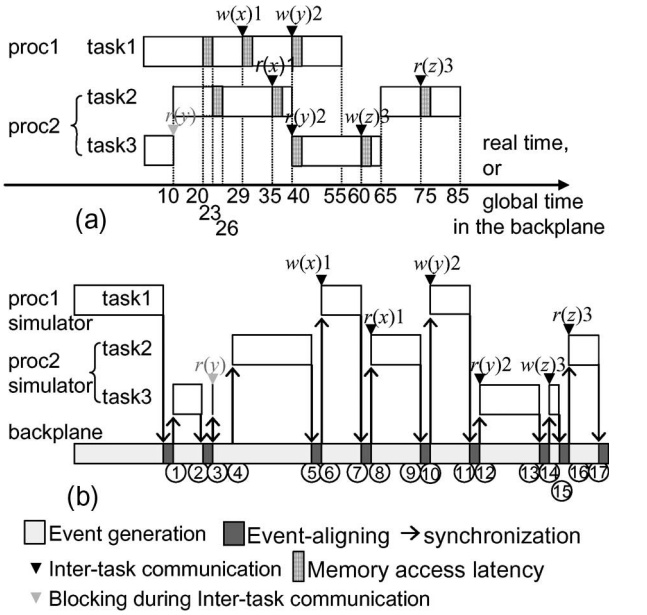


Fig. 5. Cosimulation example scenario with the proposed technique; (a) the real behavior or the result of cosimulation and (b) Execution sequence of simulators in the proposed framework.

processors; proc1 and proc2. Task1 is mapped to proc1 and task2 and task3 are mapped to proc2. We assume that task3 has a higher priority than task2 and they are scheduled by a priority-based preemptive scheduler. Intertask communication is depicted using the notation defined in Section III. For example, $w(x)1$ at time 29 means that task1 wrote a value 1 at address x. We also specify memory access latency in Fig. 5(a) assuming uniform delay for simple illustration. The actual execution sequence of component simulators by the proposed approach is shown in Fig. 5(b). The event alignment phase produces the final results as shown in Fig. 5(a), which is the same as the real behavior.

Suppose the backplane schedules proc1 simulator first. Then, task1 is executed until it encounters $w(x)1$. Before this operation, proc1 simulator returns to the backplane since a simulator has to synchronize before executing intertask communication. The backplane cannot align the events since the event queues of both task2 and task3 are empty: events of all the components in the system must be compared to advance the global clock safely. It schedules and executes task3 as it has the higher priority ($①$). Likewise, proc2 simulator returns to the backplane before $r(y)$($②$). Now that the backplane has the events of all the processor simulators, it aligns the events conservatively up to time 10 as such in Fig. 5(a). It cannot go further than time 10 as there is no trace left in the queue of task3. So it goes back to the event generation phase again.

Suppose that $r(y)$ is a blocking read operation. Task3 is blocked while performing read operation since there has been no $w(y)$ before ($③$). The proc2 simulator interface informs the backplane of the status of task3 and the OS modeler in the backplane now schedules task2 to be executed ($④$) and the proc2 simulator synchronizes before $r(x)$. The backplane can now advance the global clock again conservatively up to time 29, comparing the events of all the components in the

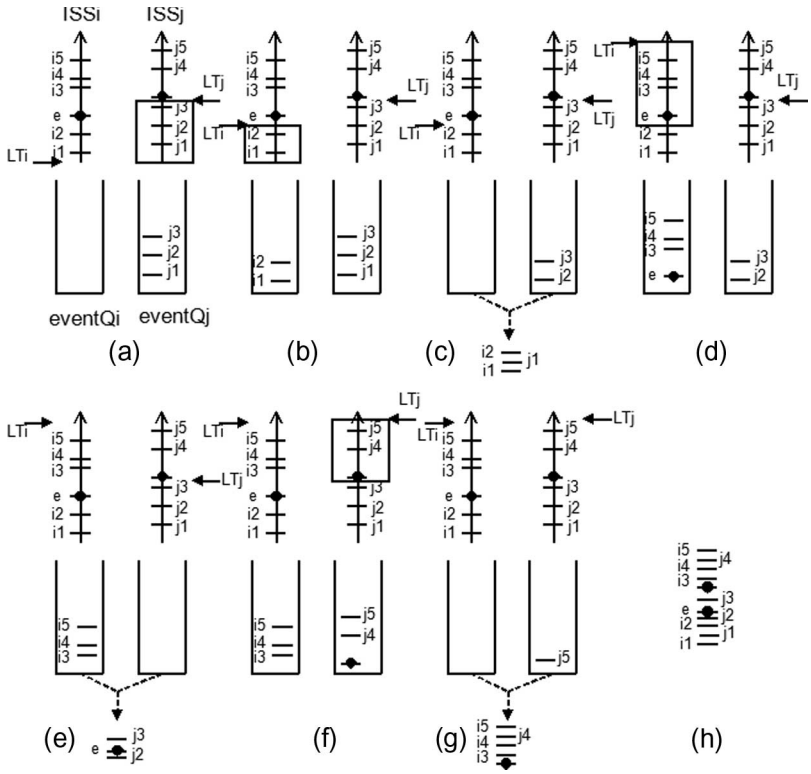


Fig. 6. Conservative alignment of events in the proposed scheme. The upper part illustrates event distribution of a simulator as its time evolves and the lower part represents the even queue of the simulator. (a) Simulator j was generating (rectangle) events and stops before e' and fills up its event queue in the backplane. (b) Since event queue of simulator i gets empty, simulator i is scheduled. (c) Events are aligned. (d) Simulator i is scheduled again and it executes e. (e) Since event queue of simulator j gets empty, simulator j is scheduled and it executes e'. (f), (g) Events are aligned. (h) Aligned events sequence.

system (⑤). Note that, while it advances the global clock, communication architecture modeler in the backplane detects that both task1 and task2 try to access the same bus at time 20. It models the contention by serializing the access according to the arbitration policy of given bus model; the memory access that is requested initially by task2 at time 20 is actually granted at time 23 and is finally completed at time 26 as such in Fig. 5(a).

In this way, the cosimulation proceeds up to time 35, repeating the two phases (⑥−⑦). Suppose that $w(y)2$ by task1 at time 40 triggers an interrupt to proc2, waking up task3 from blocking. As a result, task3 preempts task2 immediately. However, in Fig. 5(b), task2 is executed nonpreemptively until it encounters $r(z)$ and task1 completes its execution after performing $w(y)2$ (⑧−⑪). The backplane finds out that the interrupts have occurred while aligning events of task1. Event alignment pauses and the OS modeler schedules task3 to be executed (⑫). Although tasks are executed nonpreemptively in the proposed technique, the occurrence of interrupts is correctly modeled since tasks cannot execute intertask communication without first synchronizing with the backplane. Since proc2 is synchronized before $r(z)$, $w(z)3$ is executed first by task3 and $r(z)3$ is then executed by task2, performing correct cosimulation (⑬ − ⑰). Note that out-of-order simulation is performed even in the same processor simulator while preserving the causality constraints.

Compared to the previous execution-driven virtual synchronization technique, the proposed trace-driven scheme can accurately reflect the communication architecture related delay by simulating all the memory accesses trace by trace. This has the effect of synchronizing at every memory access and thus allows the backplane to detect and reflect the contention [e.g., time 20–23 in Fig. 5(a)]. In addition, the proposed scheme supports general task execution models where tasks communicate via blocking or nonblocking operations, as shown in Fig. 5(a). This has become possible by maintaining task representatives and by synchronizing at every intertask-communication as well as task execution boundaries.

B. Proof of Validity

We prove that the proposed technique satisfies the causality constraint using the definitions that we defined in Section III.

Theorem 4.1: Trace-driven virtual synchronization technique satisfies the causality constraint.

Proof: We first consider the case when there is no interrupt. According to definition 3.2, the causality constraint can be denoted as $\forall e,\ e'(e \rightarrow e' \Rightarrow \text{Execute}(e, e'))$. If such events e and e' belong to the same task, $\text{Execute}(e, e')$ is true because a simulator executes the events of the same task sequentially. Suppose that e and e' belong to the different simulators; e is executed in simulator $i(= w_i(x))$ and e' in simulator $j(= r_j(x))$ and $\text{GC}(e) < \text{GC}(e')$(①). Even if e has not yet executed and the backplane was executing simulator j, it stops before e' and executes simulator i [Fig. 6(a)] since the proposed scheme enforces any component simulator to pause before executing intertask communication and aligns events in the component event queues conservatively. Conservative alignment indicates

that, if any event queue becomes empty during the alignment, it executes the simulator and obtains the trace first before aligning traces further. The event queue of component i becomes empty [Fig. 6(c)] before that of component j since $\mathrm{GC}(e) < \mathrm{GC}(e')$. Hence, simulator i will be scheduled and execute e [Fig. 6(d)] before simulator j is scheduled and execute e' [Fig. 6(f)]. That is, $\mathrm{GC}(e) < \mathrm{GC}(e') \Rightarrow \mathrm{Execute}(e, e')(②)$. Combining $(①)$ and $(②)$ with definition 3.1 yields $(e \rightarrow e' \Rightarrow \mathrm{Execute}(e, e'))$.

Currently, let us consider the case when interrupts occur during the cosimulation. If an interrupt to a processing component is generated from the other processing component, the interrupt event is included in the generated trace. Otherwise, a separate simulation model for interrupt generator is added to the backplane. Then, the interrupt signals are aligned with other trace data in the alignment phase. When an interrupt is detected during the alignment, the OS modeler switches the execution to the interrupt handling task. ■

C. Performance and Accuracy

In addition to significantly removed time synchronization overhead, the removal of idle duration is another key advantage of the proposed technique. Since the local clock is referenced only as a timer, its absolute value is no longer meaningful. Therefore, when there is idle duration between the last execution and a new data arrival, we do not need to execute a simulator meaninglessly only to advance the local clock to the new data arrival time.

Definition 4.1 explains how trace-driven virtual synchronization technique achieves significant performance improvement. In addition to the terms defined in definition 3.4, we define the utilization of a simulator, trace-related overheads, and the total number of communication transactions to the shared memory.

Definition 4.1. Cosimulation Time With Trace-Driven Virtual Synchronization Scheme (Revised from [42]):

u_i Utilization of simulator i.

$T_{\mathrm{trans}}^{\mathrm{sh}}$ Total number of communication transactions to the shared memory. Note that T_{trans} includes $T_{\mathrm{trans}}^{\mathrm{sh}}$.

$\mathrm{st}_{\mathrm{trace}}$ Overhead per trace generation.

$\mathrm{st}_{\mathrm{eval}}$ Overhead for one trace evaluation (alignment).

$$\sum_{\forall i}^{N}\{T \times u_i \times \mathrm{st}_i\} + T_{\mathrm{trans}}^{\mathrm{sh}} \times (\mathrm{sync} + \mathrm{st}_{\mathrm{trans}}) + T_{\mathrm{trans}} \times (\mathrm{st}_{\mathrm{trace}} + \mathrm{st}_{\mathrm{eval}}). \quad (2)$$

As you can see in the (2), the total number of cycles to be simulated is reduced by $1 - u_i$ and time synchronization occurs only when data are exchanged $(T_{\mathrm{trans}}^{\mathrm{sh}})$. To clarify the cosimulation performance gain of the virtual synchronization approach over the lock-step one, (3) is obtained by subtracting (2) from (1)

$$\sum_{\forall i}\{T \times (1 - u_i) \times \mathrm{st}_i\} + \mathrm{sync} \times \left(T \times N - T_{\mathrm{trans}}^{\mathrm{sh}}\right) + \mathrm{st}_{\mathrm{trans}} \times \left(T_{\mathrm{trans}} - T_{\mathrm{trans}}^{\mathrm{sh}}\right) - T_{\mathrm{trans}} \times (\mathrm{st}_{\mathrm{trace}} + \mathrm{st}_{\mathrm{eval}}) \quad (3)$$

The positive terms are the gain and the negative ones the overheads of trace-driven virtual synchronization approach. There are mainly three kinds of gains. The first term explains the removal of idle duration of simulation. The second term shows how many synchronization points have been reduced: synchronization occurs only when data are exchanged. The third term indicates the removal of local memory transaction simulation. Instead, trace generation and evaluation overhead has been added. This overhead is insignificant and, even for DivX player application that is memory intensive, it is below 3% of total cosimulation time in our framework. Note that all the traces are contained in a memory buffer, not in a file and $\mathrm{st}_{\mathrm{eval}}$ takes place in the backplane, not in a simulator.

Accuracy of the proposed approach is affected by the accuracy of OS modeler and the communication architecture modeler. In [38], the virtual synchronization approach with OS modeler shows less than 0.1% error if cache is not used and still an acceptable level of error (less than 7%) if cache is used.

D. Consideration

The basic assumption of trace-driven simulation is that the relative time difference between events is not changed as the latency of the memory system changes. This will not hold in case of an out-of-order issue processor where instructions are scheduled dynamically, resulting in different value of relative time between events.

Another limitation is that OS itself is not executed directly on a simulator and OS modeling results in possible timing inaccuracy [38]. For example, the cache state in a processor simulator becomes different from the reality in case of preemptive scheduling since, with proposed technique, tasks are executed nonpreemptively. Suppose that a task with a long execution time gets preempted twice by another task with a short execution time and period. The proposed approach executes the long-running task nonpreemptively first and then executes the preempting task consecutively. In such a case, the tasks would experience smaller cache misses than the real situation. In other words, it may underestimate cache related preemption delay. One solution to overcome this inaccuracy is to disable the cache simulation in a processor simulator and to perform cache simulation separately in the backplane. Having a separate cache simulation in the backplane becomes necessary if one wishes to simulate a system with cache coherence protocol. The current approach assumes that intertask communication occurs only through shared memory communication.

Note that a little inaccurate timestamp of traces in event queue may result in different order of events in the system, even changing the functionality. However, such result comes from the fact that the system itself is nondeterministic or has race condition, not from the fact that the proposed time synchronization fails.

V. Enhanced Cosimulation Techniques Based on Trace-Driven Virtual Synchronization

Separation of the event generation phase and event alignment phase enables mixed level cosimulation and distributed parallel cosimulation. The former utilizes trace-based interface between

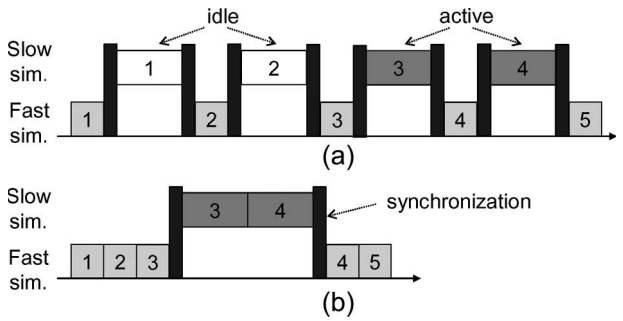


Fig. 7. (a) With lock-step synchronization, mixed level cosimulation speed is bound to the slowest simulator speed while (b) Proposed approach can maintain high cosimulation speed due to the removal of idle duration simulation and the reduced number of synchronization.

a simulator and the backplane, and the latter exploits the out-of-order simulation of the scheme.

A. Mixed Level Cosimulation With the Proposed Technique

Mixed level cosimulation is performed in order to tradeoff performance and accuracy. Moreover, mixed level cosimulation is inevitable when tasks are given in the form of fixed level IPs. In the proposed cosimulation, simulation models or simulators of different abstraction levels can be mixed together by the backplane since a simulator interacts with the backplane with the same format of traces. As long as simulators provide the required interface for trace generation, the HDL simulator and transaction level simulator can be cosimulated.

In addition, mixed level cosimulation with trace-driven virtual synchronization is fast since the overall cosimulation speed is not bound to the slowest simulator speed. A different abstraction level results in different speed of simulation. The overall cosimulation speed would be bound to the slowest simulator speed, if synchronization is performed at every cycle [Fig. 7(a)]. However, with the proposed scheme, since synchronization is performed only at intertask communication and task boundaries and idle duration is not simulated, if the portion of the simulated cycles of the slowest simulator is not dominant or the utilization of the simulator is low, the cosimulation speed would still maintain high performance [Fig. 7(b)].

In fact, we found that the proposed cosimulation is very useful to verify an RTL IP under development since real application test vectors can be fed from ISSs with fast turn around time.

We start with the description of a simulator interface to integrate a simulator into the proposed framework. The simulator interface must satisfy the following requirements.

1) It establishes the socket connection with the backplane.
2) It receives the input data from the backplane and sends the output data along with the traces to the backplane.
3) It generates memory traces in the format of ⟨address, access type, size, time difference⟩
4) It references the local clock of the simulator to measure the time difference between traces.

Related to the second requirement, the memory map of shared memory for interprocessor communication must be provided to simulator interface. This information is necessary

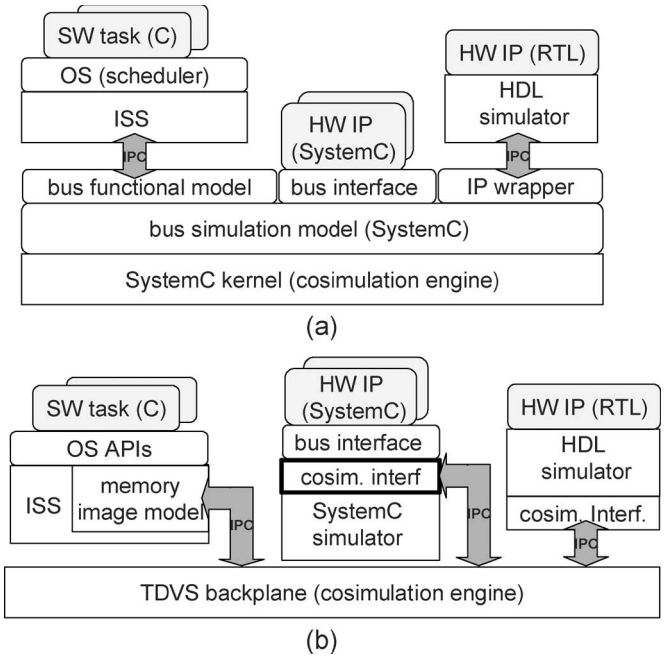


Fig. 8. (a) Conventional SystemC cosimulation framework with ISS and HDL simulator and (b) Proposed one with SystemC simulation.

for the interface to determine at run-time whether it has to exchange the data with the backplane at a certain memory access.

As for a processor simulator (ISS), it is typical to provide callback function or hooking interface for memory image in order to allow users to model their own memory-mapped peripherals. This interface is given memory access information and called every time a memory access occurs. Hence, the simulator interface is easily implemented using this hooking interface.

On the other hand, as for an HDL simulator, the interface is implemented in Foreign Language Interface (FLI) that is associated with the bus interface logic. Since the FLI module of the bus interface logic is executed every time the HW IP module tries to access memory, the interface function can be implemented. In our previous work, only the HDL simulator is integrated into the proposed framework. SystemC simulation can now be integrated as an HW simulator.

Fig. 8(a) shows the conventional SystemC cosimulation framework that includes an ISS as a processor simulator. An RTL hardware simulator is also attached to illustrate mixed level cosimulation in the conventional framework. Here, SystemC kernel plays the role of cosimulation scheduler that schedules each component simulation model (i.e., ISSs and HW IPs) and communication architecture simulation model. Note that ISS and HDL simulator are connected to the SystemC framework via interprocess communication (IPC) such as a socket. A bus functional model of ISS and IP wrapper of an HDL simulator convert memory accesses into the interface of the communication architecture simulation model.

On the contrary, Fig. 8(b) shows the proposed framework with SystemC simulator, where SystemC simulator plays the role of HW simulator, not the cosimulation scheduler. Thus, ISS and the HDL simulator must not be attached directly to the SystemC simulation framework. All the component simulators

```
void ahbmst_read(const unsigned addr, unsigned& data) {
    myCosimInterf->readData(addr, data, 1);
}
void ahbmst_write(const unsigned addr, const unsigned data) {
    myCosimInterf->writeData(addr, data, 1);
}
void cosimInterf::readData(const unsigned addr, unsigned
    data[], const int len) {
    //1. Synchronize with the backplane
    //2. Read data from the memory model
    //3. Generate trace
}
void cosimIntef::writeData(const unsigned addr, const unsigned
    data[], const int len) {
    //1. Synchronize with the backplane
    //2. Write data to the memory model
    //3. Generate trace
}
virtual void ahbslv_read(const unsigned addr, unsigned& val)=0;
virtual void ahbslv_write(const unsigned addr, const unsigned
    val)=0;
```

Fig. 9. Redefinition of bus interface to integrate SystemC simulation into the proposed cosimulation framework.

are attached to the backplane via IPC and send the generated traces. Note that bus functional model of ISS in Fig. 8(a) is replaced with the simulation interface for memory image model and an IP wrapper is replaced with the FLI module. This mitigates the effort of integrating a component simulator. Similarly, the simulator interface lies between the bus interface module and the SystemC kernel so that neither SystemC kernel nor HW IP is modified to apply the proposed scheme.

The Fig. 9 shows an example of how the simulator interface [the bold box in Fig. 8(b)] implements the interface function redefining the bus interface. The bus interface shown in the figure is a transaction level AMBA simulation model from Dynalith [43] while other general bus models can be redefined similarly. The simulator interface is implemented in cosimInterf class and there is only one instance, myCosimIntef. The AMBA master interface is redefined to call cosimInterf member functions that synchronize with the backplane. That is, it first sends the previously written data in the memory image and the generated traces via socket connection. Then, it receives acknowledgement and input data, if any, and stores them in the memory image in the simulator. It writes the data given as argument to the memory image or reads the data in the memory image to the buffer given as argument. Finally, it generates the trace of the current access that contains the information of memory access type, address, and the relative time difference to the previous trace. Simulation API such as sc_time() is used to reference the local clock.

On the other hand, the slave interface is a set of pure virtual function whose implementation is defined in IP definition and called by other masters in the bus. Thus, myCosimInterf needs to call the slave interface of an IP, if a master in the bus tries to access the address of the IP. Therefore, in the initialization phase, myCosimInterf must be given the pointer to each instance of IPs that has the slave interface.

In summary, we may increase the abstraction level of a component simulator as long as its simulation interface satisfies the interface requirements. For instance, delay annotated host code execution in SystemC may also be integrated in the mixed level cosimulation with trace-driven virtual synchronization. In such a case, the SystemC module plays the role of a high-level software simulation model.

B. Parallel Cosimulation

Since event generation can be done independently in each component simulator, the proposed technique has the potential of getting benefits from parallel simulation. Parallel execution of tasks in MPSoC architecture is achieved by mapping independent tasks in different cores or by executing tasks in a pipelined way if they have dependency. In case of mapping independent tasks in different cores, it is straightforward to execute those component simulators in parallel with the proposed scheme since there is no data dependency and component simulators do not need to synchronize at every bus access for local memory access.

However, in case there exists data dependency between tasks on different cores, parallel cosimulation of those component simulators is complicated. Parallel invocation of simulators does not necessarily lead to parallel execution of simulators. Fig. 10(a) illustrates pipelined execution in reality when a task on proc1 and a task2 on proc2 have producer–consumer dependency. Fig. 10(b) illustrates the serialized execution of simulators if synchronization is performed before every write or read operation of interprocessor communication. As in the Fig. 10(b), the backplane repeats the invocation of simulators and the waiting for the simulators. The backplane initially invokes the two component simulators concurrently. The simulator2 returns to the backplane before read operation, and so does simulator1 before write operation. At the second invocation, simulator2 becomes blocked during read operation since the buffer is empty. It cannot continue to execute until it synchronizes with the backplane and is provided with data produced by simulator1. Only at the third invocation, simulator2 can execute its first iteration and synchronizes again before the read operation of the next iteration. However, at this time, simulator1 gets blocked during write operation not knowing that simulator2 has consumed the data. In this way, the simulators are serialized.

To overcome this problem, we propose another synchronization protocol with virtual buffer. In this protocol, time synchronization is not performed before write operation of interprocessor communication but the result of write operation is kept in the virtual buffer. Virtual buffer is a representative of the pipeline buffer but whose size is enlarged enough to enable the parallel execution of simulators without blocking. Suppose that two tasks belong to the processors that correspond to pipeline stage m and stage n, respectively. Then, we increase the size of the pipeline buffer by $(1 + (n - m))$ times in the cosimulation. Due to the virtually

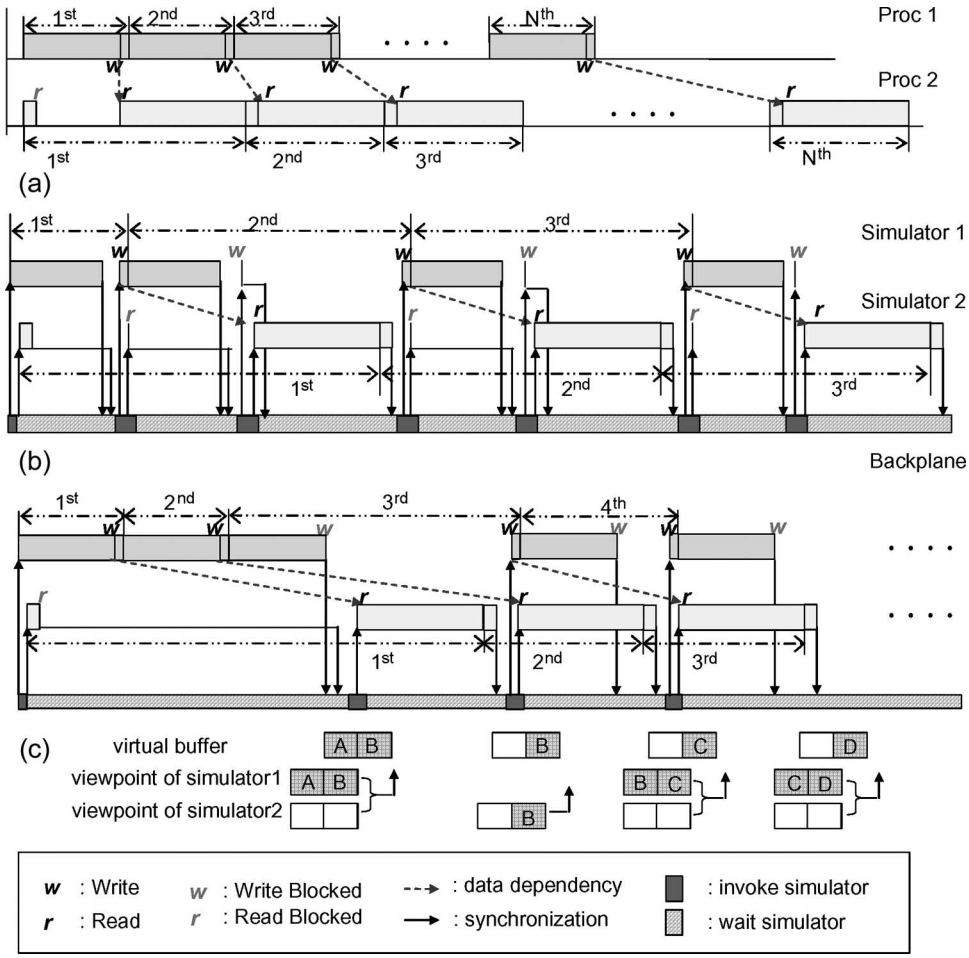


Fig. 10. (a) Pipelined parallel execution in MPSoC architecture when there are two write and read operations in each iteration, (b) Serialized execution of simulators without virtual buffer, and (c) Parallel cosimulation with virtual buffer.

increased buffer size, the simulators that execute each task do not get blocked after the initial phase, resulting in parallel cosimulation.

Fig. 10(c) shows the pipelined parallel cosimulation with virtual buffer. Since the original size of the buffer is 1 and the two processors are in the consecutive pipeline stages, the virtual buffer is set to the size of $2 = 1 + 1$. Simulator1 does not synchronize with the backplane before the write operation but it synchronizes with the backplane only if it gets blocked during the write operation. Hence, simulator1 runs at its full speed until it is blocked during the write operation in the third iteration. On the other hand, simulator2 synchronizes before the read operation in the first iteration. The backplane reconstructs the correct state of pipeline buffer in shared memory image merging the information on the buffer from each simulator. As the buffer is full (with A and B), only proc2 simulator is executed and synchronizes with the backplane before the read operation of the next iteration.

The backplane updates the state of the buffer in its shared memory reflecting the consumption (of A) by simulator2. Then, both simulators can now be resumed since the buffer is neither full nor empty. The backplane updates the production of data (of C) after simulator1 synchronizes and also updates the consumption of data (of B) after simulator2 synchronizes. As such, the virtual buffer enables parallel cosimulation.

Virtual buffer technique in a trace-driven virtual synchronization scheme is out-of-order simulation where a processor simulator that corresponds to a pipeline stage m executes the $i + 2$th iteration at the same time a processor simulator that corresponds to a pipeline stage $m + 1$ executes the ith iteration. The memory access to the increased virtual buffer is translated into the original buffer when the traces are aligned in the backplane.

We can define the cosimulation time in parallel cosimulation with trace-driven virtual synchronization as follows.

Definition 5.1. Cosimulation Time in Parallel Cosimulation With Trace-Driven Virtual Synchronization Scheme:

p_i Parallelism in simulator $i (0 < p_I < 1)$.
k Total number of simulation hosts.
N Total number of simulators.

We define the index in $T \times u_i \times p_i \times \mathrm{st}_i$ in such a way that $T \times u_1 \times p_1 \times \mathrm{st}_1$ is the largest value and $T \times u_N \times p_N \times \mathrm{st}_N$ is the smallest value

$$T \times u_1 \times p_1 \times \mathrm{st}_1 + \sum_{i=1}^{k} \{T \times u_i \times (1 - p_i) \times \mathrm{st}_i\} + \sum_{i=k+1}^{N} \{T \times u_i \times \mathrm{st}_i\} + T_{\mathrm{trans}}^{\mathrm{sh}} \times (\mathrm{sync} + \mathrm{st}_{\mathrm{trans}}) + T_{\mathrm{trans}} \times (\mathrm{st}_{\mathrm{trace}} + \mathrm{st}_{\mathrm{eval}}). \quad (4)$$

TABLE I
PARTITIONING RESULT AND THE SIMULATION TIME DISTRIBUTION OF H.263 DECODER EXAMPLE

Processor Element Name	Algorithm Block Name	Simulation Time Distribution (%)
ARM(0)	H263Reader	0.4
ARM(1)	VLD DeQuantizer InvZigZag, etc.	50.4
IDCT(Y)	IDCT_Y	1.2
IDCT(U)	IDCT_U	
IDCT(V)	IDCT_V	
ARM(2)	MotionCompensation	24.0
	DisplayFrame	24.0

The first term indicates the simulation time reduction by parallel cosimulation. Among N simulators, k simulators are executed in parallel in k simulation hosts and $T \times u_1 \times p_1 \times \mathrm{st}_1$ is chosen by the definition as $\max(T \times u_1 \times p_1 \times \mathrm{st}_1, \ldots, T \times u_k \times p_k \times \mathrm{st}_k)$. The second term indicates the portion that cannot be executed in parallel must be executed in serial. The forth and the fifth terms are identical as in (2). The third term is also the same as in (2) except the index starts from $k+1$.

VI. EXPERIMENTAL RESULTS

A. Experimental Environment and Application Example

Currently, the proposed cosimulation framework is implemented in PeaCE [44] codesign environment. We evaluated the cosimulation performance and accuracy with H.263 decoder application. As shown in Table I, Inverse Discrete Cosine Transform(IDCT) was mapped to hardware and the others were mapped to three ARM926ej-s processors. Fig. 11 shows the MPSoC architecture that we assume in this experiment, where each processor has its own local bus and local memory. Shared memory is attached to the global bus and is accessed for interprocessor communication. IDCT hardware IPs and Vectored Interrupt Controller (VIC) are also attached to the global bus. Interrupts are generated by IDCTs after consuming the input and after producing the output. Note that this example architecture is by no means optimal but assumed to be given somehow. Finding out the optimal architecture is beyond the scope of this paper.

In Table I, H263Reader reads the input file of avi format and delivers the data to the decoder frame by frame. The other blocks except DisplayFrame are executed macroblock by macroblock. Since we used the input file of QCIF format in this experiment, 99 macroblocks are decoded per frame. For each macroblock, VLD, DEQuant, InvZigzag are performed in ARM(1), IDCT in hardware IP, and MotionCompensation in ARM(2). The macroblock block decoding loop is executed in parallel on two processors and hardware IPs in a pipelined way. We performed the cosimulation until the H.263 decoder application executes three frames (I, P, P).

In the proposed cosimulation framework, ARMulator [45] was used for ARM processor simulator and IDCT was simulated in SystemC. We used 1.8 GHz Intel Dual-Xeon machine as a simulation host and target code was built using arm-elf-gcc 3.4.5 with O3 option.

B. Comparison of Cosimulation Performance and Accuracy

We compared the proposed cosimulation performance and accuracy against MaxSim, a well-known commercial SystemC cosimulation environment. The proposed framework employs trace-driven virtual synchronization and communication architecture modeling while MaxSim framework employs lock-step synchronization and provides cycle-accurate communication architecture simulation models.

For the cosimulation in MaxSim environment, H263Reader was compiled with armcc in order to use the I/O modeling support of the compiler for the file related library function such as fopen() and so on. However, the other tasks were compiled with arm-elf-gcc 3.4.5 as in the proposed cosimulation environment. The portion of H263Reader task in simulated cycles was only 1.0% (55 148 cycles). Moreover, we found that, for H263Reader, the accuracy error in simulated cycles with arm-elf-gcc 3.4.5 (03 option) was only -6% compared against one with armcc. Therefore, we ignore the error related with H263Reader task.

The result is shown in Table II. The cosimulation performance of the proposed trace-driven virtual synchronization scheme is about 300 kcycles/seven if the simulators are executed serially (VS}_serial) in one simulation server. It is more than eight times the performance improvement. As explained in the previous section, the improvement comes from the reduced time synchronization overheads, the removal of idle duration simulation, and replacing memory transaction simulation in detailed simulation model with communication architecture modeling. On the other hand, the accuracy error is defined as

$$\text{error} = \frac{\text{simulatedCycles}_{\text{proposed}} - \text{simulatedCycles}_{\text{reference}}}{\text{simulatedCycles}_{\text{reference}}}$$

and is about 5% for this example. The error arises from the architecture modeling error as explained in the previous section and from the little difference in the two target codes for different cosimulation frameworks.

If the simulators are executed in parallel with the virtual buffer (VS}_parallel), the performance is about 400 kcycles/s, resulting in the 11 times faster speed than the reference framework. The performance improvement by parallel cosimulation against the serial cosimulation in the proposed framework is about 28% for this example. The gain can vary depending on the parallelism in the partitioned application. Two processors and IDCTs are executed in a pipelined way consisting of three stages on the dual core machine. Therefore, we can estimate the upper bound of performance improvement by parallel cosimulation as $100/\{\max(50.4, 24.0 + 1.2) + 0.4 + 24.0\} = 1.34$, where the distribution of simulation time is obtained by executing the application serially as shown in Table I. Note that since H263 Reader task and DisplayFrame block are executed frame by frame, they cannot be executed in the pipelined loop of macroblock decoding. The difference in accuracy error between VS}_serial and VS}_parallel comes from the little difference in the target codes due to virtual buffer.

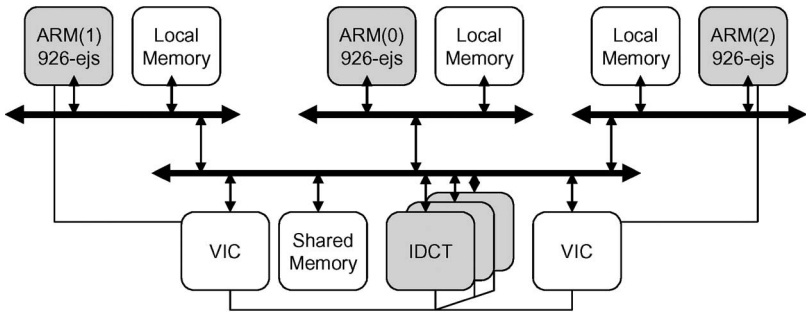


Fig. 11. MPSoC architecture we assume in H.263 decoder example.

TABLE II
PERFORMANCE AND ACCURACY COMPARISON OF THE PROPOSED FRAMEWORK WITH MAXSIM

	Simulated Cycles	Simulation Time	Speed	Error	Speedup
MaxSim 6.0	5,053,686 cycles	142.57 sec	35,447 cycles/sec	0%	1.00
VS_serial	5,329,886 cycles	17.20 sec	309,877 cycles/sec	5%	8.74
VS_parallel	5,261,980 cycles	13.24 sec	397,430 cycles/sec	4%	11.21

TABLE III
PARTITIONING RESULT AND THE SIMULATION TIME DISTRIBUTION OF DIVX PLAYER EXAMPLE

Processor Element Nam	Algorithm Block Name	Simulation Time Distribution (%)
ARM(0)	MP3 decoder	2.4
	Avi Parser	0.2
	VLD, etc	15.2
ARM(1)	DeQuantizer	20.7
ARM(2)	IDCT, etc	29.5
ARM(3)	MotionCompensation, etc	16.0
	DisplayFrame	16.0

TABLE IV
DISTRIBUTED PARALLEL COSIMULATION OF DIVX PLAYER EXAMPLE

	Simulation Time	Speedup
VS serial	24.37 sec	1.00
VS parallel with 4 simulation hosts	11.87 sec	2.05

We applied the proposed technique to the different application in order to show that the proposed parallel cosimulation can be performed as well in distributed simulation hosts. Table III shows the mapping results of DivX player into four ARM processor simulators. DivX player consists of H.263 decoder, mp3 decoder, and avi file parser. The cosimulation was performed in parallel in four distributed simulation hosts that has 3.0 GHz Intel PentiumD processor and each ARM processor simulator was executed in each simulation host. The upper bound of the performance is estimated as $100/\{\max(16.0, 29.5, 20.7, 15.2) + 16.0 + 2.4 + 0.2\} = 2.08$. We obtained the 2.05 times of performance improvement as shown in Table IV.

The result illustrates that, with the proposed parallel cosimulation scheme, performance gain increases as the number of processor simulators does, just like in the real MPSoC architecture.

TABLE V
MIXED LEVEL COSIMULATION OF H.263 DECODER EXAMPLE

	Simulation Time (portion)	Speed
RTL (ModelSim)	54.13 sec (71.50%)	2, 177 cycles/sec
SystemC	0.25 sec (0.33%)	470,448 cycles/sec
ISS (ARMulator)	19.10 sec (25.23%)	222,206 cycles/sec
Backplane	2.23 sec (2.95%)	N/A
Overall	75.71 sec (100.00%)	70,883 cycles/sec

Finally, Table V shows the results of the mixed level cosimulation where two different abstraction levels of hardware simulators are cosimulated with an ISS. Both MotionCompensation and IDCT blocks are mapped to hardware but MotionCompensation is modeled in RTL and simulated in modelSim while IDCTs are modeled in TLM and simulated in SystemC simulator. The other blocks and tasks are mapped to an arm926ej-s core and simulated on the ARMulator. In the mixed level cosimulation with the proposed scheme, although HDL simulator is the bottleneck of the overall performance, the overall performance is not bound to the performance of HDL simulator. The overall cosimulation performance is still over 70 kcycles/s in this example.

VII. CONCLUSION

In this paper, we have proposed a novel HW/SW cosimulation technique which enables fast and accurate hardware/software cosimulation for MPSoC. Trace-driven virtual synchronization scheme boosts the speed of cosimulation by reducing the time synchronization overhead to almost zero, while it considers dynamic behavior of the system such as OS scheduling and contention in the bus. We have proved the validity of the proposed time synchronization scheme and

analyzed the performance gain factors of the proposed scheme against the lock-step synchronization one.

Two enhanced cosimulation techniques that utilize the proposed time synchronization scheme are also presented; one is a fast mixed level cosimulation where SystemC simulation is seamlessly integrated and the overall cosimulation speed is not bound to the speed of the lowest abstraction level simulator. The other is parallel cosimulation technique for a pipelined parallel execution where out-of-order parallel cosimulation is performed on distributed simulation hosts or different cores in a multicore machine. Through parallel SystemC cosimulation, it is shown that the simulation speed can be boosted up as the real system does with parallel processors.

Compared with MaxSim, a well-known SystemC cosimulation framework, the proposed framework showed 11 times faster cosimulation speed while the error was maintained below 5% for H.263 decoder example.

ACKNOWLEDGMENT

The ICT and ISRC at Seoul National University and IDEC provided research facilities for this study.

REFERENCES

[1] W. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, vol. 82, no. 7, pp. 967–989, Jul. 1994.

[2] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the multi-processor SoC design space with SystemC," *J. VLSI Signal Process.*, vol. 41, no. 2, pp. 169–182, Sep. 2005.

[3] ARM Ltd., *RealView MaxSim*. [Online]. Available: http://www.arm.com/products/DevTools/MaxSim.html

[4] C. Valderrama, F. Nacabal, P. Paulin, and A. Jerraya, "Automatic generation of interfaces for distributed C-VHDL cosimulation of embedded systems: An industrial experience," in *Proc. Rapid Syst. Prototyp.*, 1996, pp. 72–77.

[5] D. Becker, R. Singh, and S. Tell, "An engineering environment for hardware/software co-simulation," in *Proc. Des. Autom. Conf.*, Jun. 1992, pp. 129–134.

[6] W. Sung and S. Ha, "A hardware software cosimulation backplane with automatic interface generation," in *Proc. Asia South Pac. Des. Autom. Conf.*, 1998, pp. 177–182.

[7] C. Passerone, L. Lavagno, C. Sansoe, M. Chiodo, and A. Sangiovanni-Vincentelli, "Trade-off evaluation in embedded system design via co-simulation," in *Proc. Asia and South Pac. Des. Autom. Conf.*, Jan. 1997, pp. 291–297.

[8] C. Passerone, L. Lavagno, M. Chiodo, and A. Sangiovanni-Vincentelli, "Fast hardware/software co-simulation for virtual prototyping and trade-off analysis," in *Proc. Des. Autom. Conf.*, Jun. 1997, pp. 389–394.

[9] V. Zivojnovic and H. Meyr, "Compiled HW/SW co-simulation," in *Proc. Des. Autom. Conf.*, Jun. 1996, pp. 690–695.

[10] B. Tabbara, M. Sgroi, A. Sangiovanni-Vincentelli, E. Filippi, and L. Lavagno, "Fast hardware-software co-simulation using VHDL models," in *Proc. Des. Autom. Test Eur.*, Mar. 1999, pp. 309–316.

[11] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, 2002.

[12] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proc. Hardware/Software Codes. Syst. Synth.*, Oct. 2003, pp. 19–24.

[13] L. Semeria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++," in *Proc. Asia South Pac. Des. Autom. Conf.*, 2000, pp. 405–408.

[14] A. Bouchhima, S. Yoo, and A. Jerraya, "Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model," in *Proc. Asia and South Pac. Des. Autom. Conf.*, 2004, pp. 469–474.

[15] CoWare Inc., *ConvergenSC*. [Online]. Available: http://www.coware.com/products/

[16] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC cosimulation and emulation of multiprocessor SoC designs," *Computer*, vol. 36, no. 4, pp. 53–59, Apr. 2003.

[17] F. Fummi, S. Martini, G. Perbellini, and M. Poncino, "Native ISS-SystemC integration for the co-simulation of multi-processor SoC," in *Proc. Des. Autom. Test Eur.*, Mar. 2004, pp. 564–569.

[18] L. Formaggio, F. Fummi, and G. Pravadelli, "A timing-accurate HW/SW co-simulation of an ISS with SystemC," in *Proc. Hardware/Software Codes. Syst. Synth.*, Sep. 2004, pp. 152–157.

[19] F. Fummi, M. Loghi, S. Martini, M. Monguzzi, G. Perbellini, and M. Poncino, "Virtual hardware prototyping through timed hardware-software co-simulation," in *Proc. Des. Autom. Test Eur.*, Mar. 2005, pp. 798–803.

[20] P. Gerin, S. Yoo, G. Nicolescu, and A. Jerraya, "Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures," in *Proc. Asia and South Pac. Des. Autom. Conf.*, Jul. 2001, pp. 63–68.

[21] A. Sayinta, G. Canverdi, M. Pauwels, A. Alshawa, and W. Dehaene, "A mixed abstraction level co-simulation case study using SystemC for system-on-chip verification," in *Proc. Des. Autom. Test Eur.*, Mar. 2003, pp. 95–100.

[22] R. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[23] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, vol. 24, no. 11, pp. 198–206, Apr. 1981.

[24] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[25] W. Sung and S. Ha, "Efficient and flexible cosimulation environment for DSP applications," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.—Special Issue VLSI Design CAD Algorithms*, vol. E81-A, no. 12, pp. 2605–2611, Dec. 1998.

[26] J. Jung, S. Yoo, and K. Choi, "Performance improvement of multi-processor systems cosimulation based on SW analysis," in *Proc. Des. Autom. Test Eur.*, Mar. 2001, pp. 749–753.

[27] M. Chung and C. Kyung, "Enhancing performance of HW/SW cosimulation and coemulation by reducing communication overhead," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 125–136, Feb. 2006.

[28] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. Hill, J. Larus, and D. Wood, "Fast and portable parallel architecture simulators: Wisconsin wind tunnel II," *IEEE Concurrency*, vol. 8, no. 4, pp. 12–20, Oct.–Dec. 2000.

[29] S. Yoo and K. Choi, "Optimistic distributed timed cosimulation based on thread simulation model," in *Proc. Int. Workshop Hardware/Software Codes.*, Mar. 1998, pp. 71–75.

[30] K. Hines and G. Borriello, "Dynamic communication models in embedded system co-simulation," in *Proc. Des. Autom. Conf.*, Jun. 1997, pp. 395–400.

[31] Mentor Graphics, Inc., *SeamlessCVE*. [Online]. Available: http://www.mentorg.com/seamless

[32] R. Uhlig and T. Mudge, "Trace-driven memory simulation: A survey," *ACM Comput. Surv.*, vol. 29, no. 2, pp. 128–170, Jun. 1997.

[33] S. Goldschmidt and J. Hennessy, "The accuracy of trace-driven simulations of multiprocessors," in *Proc. Meas. Model. Comput. Syst.*, Jun. 1993, pp. 146–157.

[34] F. Terpstra, S. Polstra, A. Pimentel, and B. Hertzberger, "Rapid evaluation of instantiations of embedded systems architectures: A case study," in *Proc. Progress Workshop Embed. Syst.*, Oct. 2001, pp. 251–260.

[35] E. Kock, G. Essink, W. Smits, P. Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers, "Yapi: Application modeling for signal processing systems," in *Proc. Des. Autom. Conf.*, Jun. 2000, pp. 402–405.

[36] D. Kim, C. Rhee, and S. Ha, "Combined data-driven and event-driven scheduling technique for fast distributed cosimulation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 5, pp. 672–678, Oct. 2002.

[37] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 4th ed. Reading, MA: Addison-Wesley, 2005, pp. 755–759.

[38] Y. Yi, D. Kim, and S. Ha, "Fast and time-accurate cosimulation with OS scheduler modeling," *Des. Autom. Embed. Syst.*, vol. 8, no. 2/3, pp. 211–228, Sep. 2003.

[39] T. Oh, Y. Yi, and S. Ha, "Communication architecture simulation on the virtual synchronization framework," in *Proc. Int. Workshop Syst., Archit., Model. Simul.*, Jul. 2007, pp. 1–10.

[40] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proc. Des. Autom. Conf.*, Jun. 2004, pp. 113–118.

[41] G. Schirner and R. Domer, "Accurate yet fast modeling of real-time communication," in *Proc. Hardware/Software Codes. Syst. Synth.*, Oct. 2006, pp. 70–75.

[42] D. Kim, Y. Yi, and S. Ha, "Trace-driven HW/SW cosimulation using virtual synchronization technique," in *Proc. Des. Autom. Conf.*, Jun. 2005, pp. 345–348.
[43] *Dynalith*. [Online]. Available: http://www.dynalith.com
[44] S. Ha, C. Lee, Y. Yi, S. Kwon, and Y. Joo, "Hardware-software codesign of multimedia embedded systems: The PeaCE approach," in *Proc. Embed. Real-Time Comput. Syst. Appl.*, Aug. 2006, vol. 1, pp. 207–214.
[45] ARM Ltdl, *RealView ARMulator*. [Online]. Available: http://www.arm.com/products/DevTools/RealViewDevSuite.html
[46] Mentor Graphics, Inc., *ModelSim*. [Online]. Available: http://www.mentor.com/products/fv/digital_verification/modelsim_se/index.cfm

Youngmin Yi (M'07) received the B.S. degree in computer engineering and the M.S. and Ph.D. degrees in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 2000, 2002, and 2007, respectively.

He was a Research Fellow with the Embedded Software Institute, Korea University, Seoul. He is currently a Postdoctoral Researcher with the University of California, Berkeley. His research interest includes hardware/software codesign, system-level simulation, and embedded software design for multiprocessor system-on-chip.

Dohyung Kim received the B.S. degree in computer engineering and the M.S. and Ph.D. degrees in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 1997, 1999, and 2004, respectively.

He was a Postdoctoral Researcher with Seoul National University, in 2004, and with the University of California at San Diego, La Jolla, from 2005 to 2006. He is currently a Software Engineer with Google, Inc., Mountain View, CA, starting from 2007. His research interest includes various aspects of multiprocessor systems such as specification, performance prediction and simulation.

Soonhoi Ha (S'87–M'94) received the B.S. and M.S. degrees in electronics engineering from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively, and the Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley, Berkeley, in 1992.

He was with Hyundai Electronics Industries Corporation, Seoul, from 1993 to 1994 before he joined the faculty of the School of Electrical Engineering and Computer Science, Seoul National University, where he is currently a Professor. He is a Program Cochair of CODES+ISSS'2006, ASPDAC'2008, and ESTIMedia'2005–2006. He has been a member of the technical program committee of several technical conferences including DATE, CODES+ISSS, and ASP-DAC. His primary research interests are various aspects of embedded system design including hardware/software codesign, design methodologies, and embedded software design for multiprocessor system-on-chip.