Virtual Synchronization Technique with OS Modeling for Fast and Time-accurate Cosimulation

Youngmin Yi Dohyung Kim Soonhoi Ha

CAP Laboratory
Department of Computer Science and Engineering
Seoul National Univ., Seoul, Korea
+82 2 8807292

{ymyi, dhkim, sha}@iris.snu.ac.kr

ABSTRACT

Hardware/Software cosimulation is the key process to shorten the design turn around time. We have proposed a novel technique, called virtual synchronization, for fast and time accurate cosimulation that involves interacting component simulators. In this paper, we further extend the virtual synchronization technique with OS modeling for the case where multiple software tasks are executed under the supervision of a real-time operating system. The OS modeler models the RTOS overheads of context switching and tick interrupt handling as well as preemption behavior. While maintaining the timing accuracy to an acceptable level below a few percents, we could reduce the simulation time drastically compared with existent conservative approach by removing the need of time synchronization between simulators. It is confirmed with a preliminary experiment with a multimedia example that consists of four real-life tasks.

Categories and Subject Descriptors

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

General Terms

Performance, Verification.

Keywords

Cosimulation, virtual synchronization, OS modeling

1. INTRODUCTION

Software modules are taking more and more roles in embedded systems and it is crucial to enable software development concurrently with the hardware development to reduce the design time. This can be achieved by hardware/software cosimulation in the hardware/ software codesign methodology that uses a virtual prototype.

Cosimulation can be used to evaluate several candidate architectures in the architecture selection stage of which the goal is to find out an optimal architecture for a given application. A popular cosimulation technique in this stage is to use host code execution with delay annotation. A task is not cross-compiled to target code but compiled and executed as host code and its timing information is estimated a priori and annotated. Data exchange between processing components is modeled and performed at the transaction level. Since static timing estimation cannot take into account any run-time variation of performance, timing accuracy of cosimulation is limited. Recently, there have appeared many research results ([3][4][6]) and commercial tools([14][15]) that are considering this level of abstraction.

Cosimulation can also be used for coverification of designed system where timing accuracy is the key requirement. In this stage, time-accurate instruction set simulator (ISS) and RTL simulator are usually used for software component and for hardware component respectively[1][14][15]. As a result, cosimulation involves multiple component simulators running concurrently and interacting with each other. It is known that time-accurate cosimulation is much slower than delay-annotated transaction level cosimulation by some orders of magnitude. Since a time-accurate cosimulation may also be used for the last stage of design space exploration such as fine-tuning of communication architecture and memory systems, it is desirable to speed up the cosimulation speed, which is the main concern of this paper.

There are two main causes for low performance of time accurate cosimuation: one is slow simulation speed of each component simulator, and the other is time synchronization overhead between component simulators. There have been several efforts to speed up time-accurate cosimulation. Compiled ISS[2] was proposed to boost the simulation speed of software simulator. It achieves high simulation speed by removing the overhead of instruction decoding at run time. Instead, it translates each target instruction directly to one or more host instructions at compile time. But it has a drawback that it lacks of adaptability to the modification of the architecture or the compiler. Also, it is difficult to apply this approach to the dynamic program such as OS model. Moreover, time synchronization overhead remains between component simulators.

Recently, we have proposed the virtual synchronization technique for distributed time-accurate cosimulation to reduce the simulation time and the time synchronization overhead simultaneously[7]. The virtual synchronization technique eliminates the need for time synchronization between component simulators at all: synchronization appears accomplished only when events are exchanged. Also it improves the performance of an individual simulator by removing the overhead of redundant local clock advancement.

However, there is a critical constraint on the simulated tasks to apply the virtual synchronization technique. The task execution

CODES+ISSS '03, October 1-3, 2003, Newport Beach, California, USA

model that virtual synchronization assumes is that the execution results of the task do not depend on the arrival times of input events but only on the arrival order of them. Once a task is executed with a given set of input samples, it assumes completed without interruption.

In this paper, we extend the virtual synchronization technique to the case where multiple software tasks are executed under the supervision of a real-time operating system (RTOS) in a processor. Virtual synchronization cannot be directly applied in this case since an RTOS does not satisfy the execution model that virtual synchronization assumes: if the input samples for a task with higher priority arrives at the processor, the RTOS preempts the current task execution in the middle. On the other hand, it is a big burden to run the RTOS itself on the processor ISS. The proposed approach runs only application tasks on the ISS and models the RTOS in the cosimulation backplane to achieve faster cosimulation still preserving the timing accuracy to an acceptable level below a few percents.

In the next section, we explain the time synchronization problem between component simulators and the virtual synchronization technique our work is based on. Section 3 presents motivation and section 4 describes the proposed technique with RTOS modeling. In section 5, we formulate the expected performance improvement when the modeling is used. Experimental results are shown in section 6. Related work and conclusions follow in section 7 and 8 respectively.

2. TIME SYNCHRONIZATION PROBLEM AND VIRTUAL SYNCHRONIZATION

A time-accurate cosimulation environment consists of a set of component simulators. Figure 1(a) illustrates a simple example that involves a HW simulator and a SW simulator. The source block models an environment and generates the triggering events to the SW task periodically with period of 8 time units. The number denoted inside a block indicates the simulated time duration of event processing: for example, the SW task takes 4 time units for event processing. And we assume for simple illustration that the SW simulator is twice faster than the HW simulator to advance its local clock one time unit.

When multiple simulators are involved, the main difficulty is to synchronize the simulators to avoid the causality error. A simple but most popular scheme for time synchronization between component simulators is conservative approach[11]. In this approach, the local clock advancement is marshaled at every time unit as illustrated in Figure 1(b). Then, the cosimulation speed is bounded by the performance of the slowest simulator further degraded by synchronization overhead. And the component simulator takes simulation time for clock advancement without doing any useful work during the idle time. To reduce the synchronization overhead, some advanced techniques have been developed: optimized approach[13] and optimistic approach[12]. Unfortunately, they usually assume special features of component simulators so their applicability is severely restricted. Virtual synchronization technique, on the other hand, does not assume such special features of simulators. Instead, it assumes the execution model of simulated tasks: they are functional and monotonic meaning that the output results of the tasks depend only on the arrival order not on the arrival times of input events.

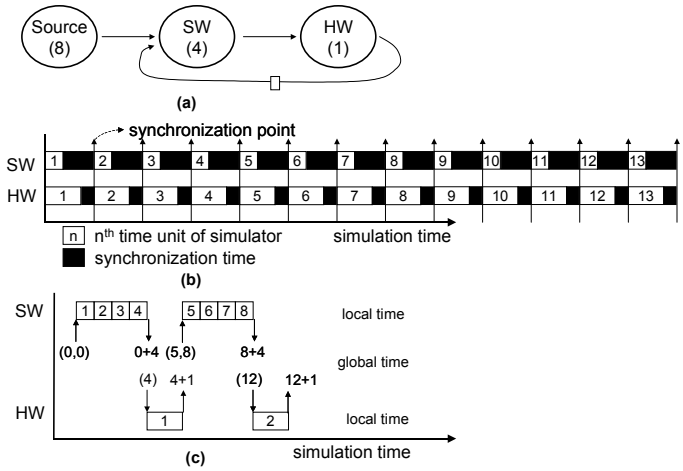

Figure 1. (a) A simple cosimulation example, (b) cosimulation scenario of a conservative approach, (c) that of the virtual synchronization approach.

A typical software function meets this requirement. Suppose a software task is a function call with two arguments of which one is supplied by the source block and the other by the hardware block in Figure 1(a). Then, the execution of this task waits until two arguments arrive at the input ports. At the second invocation of the software block, one event arrives at time 8 from the source block and the other at time 5 from the hardware block. Then, the software block starts execution at time 8 and produces an output event at time 12. It means that the software block does no useful work from simulated time 4 to time 8.

In virtual synchronization, the local clock of a component simulator is not synchronized with the system-wide global clock. Instead, a wrapper is inserted at the interface between the component simulator and the cosimulation backplane as shown in Figure 2(a). When a data sample d1 is delivered from the backplane to the simulator at simulated time t1, the wrapper translates the time stamp of the event to the current time of the component simulator. When it receives an output event from the simulator, it computes the elapsed time Δ(d1) and reconstructs the correct time stamp of the output event with t1 + Δ(d1).

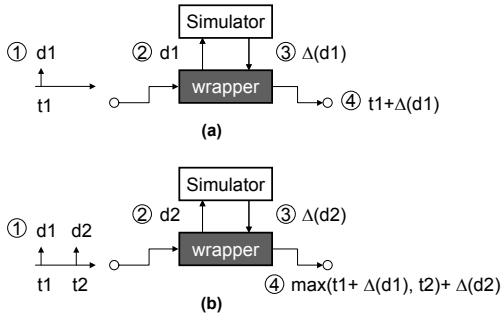

Figure 2. Cosimulation wrapper is inserted for virtual synchronization to translate the time stamps of the input and the output events.

Figure 2(b) illustrates how the wrapper adjusts the time stamp of the next event assuming that input events d1 and d2 arrive at t1 and t2 respectively. If t2 is larger than t1+Δ(d1), the output event for d2 simply has the time stamp t2+Δ(d2). Special care should be

taken when t2 is smaller than t1+Δ(d1) meaning that the next input event arrives while the component processes the current event. Since a software simulator may accept the input event after completion of the current execution. Then, the time stamp of the output event would be t1+Δ(d1) + Δ(d2).

Thus, time synchronization is accomplished when data samples are exchanged. It means that virtual synchronization removes all time synchronization overheads except the time stamp translation overhead of the wrapper, which is negligible. Figure 1(c) illustrates the scenario when the virtual synchronization technique is used for the cosimulation of Figure 1(a). The software simulator clock stops advancement at time 4 after sending an output event to the hardware simulator. The hardware simulator receives the input event at local time 0 and advances 1 time unit to produce an output. Therefore, the wrapper of the hardware simulator translates the time stamp of the input event from 4 to 0 and reconstructs the time stamp of the output event from 1 to 5. The software simulator resumes the second invocation after it receives an event from the source block at simulated time 8.

Note that virtual synchronization not only removes the synchronization overhead but also reduces the active duration of component simulators. A simulator does not need to increase the local clock until it receives a new input data after processing the last data samples. Thus, the virtual synchronization technique boosts the performance of the simulator itself.

3. MOTIVATION AND OS MODELER

While the virtual synchronization technique requires that a software task runs to completion once invoked, it is not the case if the task runs under the supervision of an operating system. A real time operating system (RTOS) usually services a periodic tick interrupt for real-time scheduling based on accurate time duration in terms of real-time clock ticks. It may preempt the current task execution with later arrived input samples if they trigger a task with a higher priority. If there are multiple runnable tasks at the same time, which task would be executed depends on the scheduling policy of the RTOS.

Figure 3 illustrates an example case where a G.723 decoder task and an H.263 decoder task are executed in the same software component. Figure 3(a) shows the desired simulation behavior when the G.723 decoder has a higher priority than the H.263 decoder and an input stream to the G.723 decoder arrives later. Since virtual synchronization executes the H.263 decoder first without interruption as shown in Figure 3(b), it fails to simulate the preemption behavior.

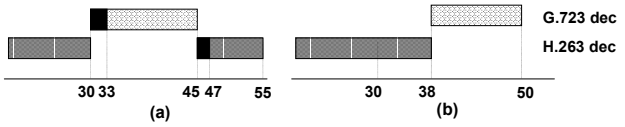

Figure 3. Executing multiple tasks on a simulator: G.723 decoder has higher priority than H.263 decoder. (a) Correct execution scenario, and (b) execution scenarios with virtual synchronization of the previous section.

On the other hand, it is a big burden to run the RTOS itself on the processor ISS, which we want to avoid for faster cosimulation. Executing the applications with an RTOS involves execution of RTOS kernel codes such as scheduler codes as well as context switching routines. In addition, it also involves the execution of idle tasks when CPU is in the idle state. In fact, during an idle period, an RTOS generally runs an idle task that is merely a loop without doing any meaningful work.

Another disadvantage of running the RTOS on the processor ISS is that tick interrupt modeling is required to advance the global OS tick for task scheduling. Most simulators supply the means to model the periodic exception generator. However, we observe that the use of this kind of simulation API degrades the overall simulator performance. This is because the simulator must check at every simulated cycle whether the scheduled (booked) exception generator should be invoked or not.

Therefore, for fast cosimulation, we want to avoid running the RTOS itself but run only the application tasks on the ISS. The question is how to reconstruct the correct time stamp of the output events from the software simulator considering the effects of the RTOS supervision. The proposed solution is to model the effect of the RTOS on the task execution time in the wrapper: we define "OS modeler" to perform this job. An OS modeler that resides in the wrapper of the software component simulator plays the role of adjusting the time stamps considering the RTOS scheduling policy.

In the proposed technique, we execute a software task ignoring the preemption possibility using the virtual synchronization technique. As shown in Figure 3(b), H.263 decoder ends the execution at time unit 38 and returns the output data to the associated OS modeler. The OS modeler waits until it receives the next input data from the backplane that will be sent to the simulator. When it receives the input data for the G.723 decoder at time 30, it compares the time stamp with that of the H.263 decoder output data. Since the G.723 decoder has a higher priority and the time stamp of its input data is earlier than that of the H.263 decoder output, the OS modeler postpones setting the time stamp of the output data until the G.723 decoder completes. After it completes, the OS modeler marks the time stamp of the output data considering the effect of preemption and the OS overhead. The time stamp of the output data of the H.263 decoder is finally set to be 55 as shown in Figure 3(a). Such adjustment mimics the preemption behavior successfully without preemption in reality. This approach achieves faster cosimulation, still preserving timing accuracy to an acceptable level below a few percents. Detailed OS modeling is the theme of the next section.

4. PROPOSED RTOS MODELING

An OS modeler models not only the preemption behavior but also the RTOS overhead during task execution. Two main ingredients of the RTOS overhead are the context switch overhead and the tick interrupt handling overhead. Whenever a tick interrupt (timer interrupt) occurs, the current task is preempted and the tick interrupt handler is called. The handler clears the interrupt and calls the scheduler. The scheduler of RTOS examines whether there exists any ready (runnable) task that has a higher priority than the current task. If this is the case, it calls the subsequent context switching routine that saves the context of the current task and restores the context of the newly scheduled task. Otherwise, no further routine is called and the RTOS yields the control to make the current task resume its execution. We assume that other interrupt handling tasks are modeled as separate software tasks that have higher priority than application tasks. To model the RTOS scheduler accurately, therefore, the OS modeler should be

informed of the accurate context switch overhead and the execution time of the tick interrupt handler. With a given frequency of tick interrupt, the OS modeler can accurately calculate how many interrupts occurs during task execution.

The OS modeler determines the response time, R_i , of a software task by the following formula for the priority-based preemptive scheduler:

$$\overline{E}_i = f(E_i) \cdot E_t + E_i \quad , \quad R_i = \overline{E}_i + \sum_{\forall j, \pi_j > \pi_i, S_i \le S_j \le S_i + R_i} (\overline{E}_j + C)$$

where, $f(x) = \left\lfloor \frac{x}{tick\ size - E_t} \right\rfloor$ is the number of tick interrupt occurrences during x. Tick size is the time duration between the consecutive timer interrupts. E_t is the execution time of the tick interrupt handler. E_i is the execution time, s_i is the start time, and π_i is the priority of task i. C is the context switch overhead. The second term of the formula is the total preemption time and the resultant context switch overhead that task i suffers by those tasks with higher priority. The accuracy of the proposed OS modeling approach depends on the accuracy of the estimated overheads of context switch and tick interrupt handling. To guarantee predictable timing behavior, an RTOS usually has the constant overheads or at least bounded overheads. Therefore, the proposed formula provides very accurate results as demonstrated in the experiments.

A major source of inaccuracy is cache. If cache is used, the estimated response time of a task from the proposed approach might be different from the real one. Consider a situation illustrated in Figure 4. Figure 4(a) and 4(b) describes the actual task execution order and a possible task execution order in the proposed scheme respectively. If the G.723 decoder is executed consecutively as in Figure 4(b), it experiences smaller cache misses than the real situation. It can shorten the estimated execution time of the second instance of G.723. Moreover, the G.723 decoder pollutes the cache states of the H.263 decoder task, which cannot be modeled in the proposed approach.

Although the OS modeler approach cannot model these types of cache misses, experiments show that the resultant inaccuracy is negligible for most of the multimedia applications, which is the main target application of our concern. Since multimedia applications tend to have large size data of stream type, temporal locality is less significant than spatial locality. The proposed scheme fails to model accurately the effect of temporal locality.

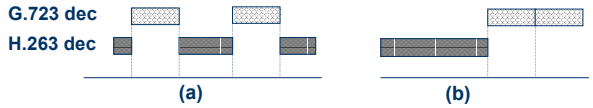

Figure 4. Task execution order (a) in reality and (b) in the proposed approach before time adjustment

Finally, the proposed OS modeler enables us to choose an optimal RTOS and its scheduling policy as well as task priorities without modifying the real RTOS. Therefore, fast and time accurate cosimulation with OS modeling can be effectively used for the last stage of design space exploration: OS selection and optimization.

5. EXPECTED PERFORMANCE IMPROVEMENT

In this section, we compute the expected performance improvement from the proposed technique of RTOS modeling against running the RTOS code directly on the ISS. The overheads of running the RTOS code are three-folds. First, there is a run-time overhead of kernel mode associated with tick interrupt handling and context switching. We denote r as the ratio of this overhead to the total simulation time. If the tick interrupt overhead dominates the context switch overhead, as usually is the case, this overhead is nearly equal to the ratio of the execution time of the tick interrupt handler to the tick size. Second, the RTOS runs a default idle task when there is no useful work to do. Therefore, there is no visible idling duration during the simulation. On the other hand, we can save this idle duration if we use the virtual synchronization technique only to the application code. We denote i as the ratio of the CPU idle duration to the total simulation time. Third, we have to call a specific API provided in the simulator to schedule (or book) the exception generator periodically inside the simulator. We denote s as the degradation ratio of the simulation time to advance one simulated time unit.

Now, we further define the following notations to establish a formula of the expected performance improvement:

ST_{RTOS} : the total simulation time in case we run the RTOS on the ISS

ST_{model} : the total simulation time in case we execute the application tasks only through the proposed scheme

T : the simulation time of the application tasks

R : the simulation time for the RTOS overheads

I : the simulation time for the idle task execution

Then, the performance improvement would be $1 - \frac{ST_{model}}{ST_{RTOS}}$.

Since $ST_{RTOS} = T + R + I$ and $ST_{model} = \frac{T}{(1+s)}$, the performance improvement can be expressed as

$$1 - \frac{ST_{model}}{ST_{RTOS}} = 1 - \frac{\frac{T}{(1+s)}}{T + R + I} \qquad (5.1)$$

Since $R = (T + R + I) \cdot r$ and $I = (T + R + I) \cdot i$, equation (5.1) is reduced to

$$1 - \frac{ST_{model}}{ST_{RTOS}} = 1 - \frac{1 - (r + i)}{1 + s} = \frac{s + r + i}{1 + s} \qquad (5.2)$$

Equation (5.2) indicates the expected performance gain we can get from the proposed approach without taking the synchronization overhead into account. If we consider the synchronization overhead, the expected performance is further multiplied by the gain due to the removal of the synchronization overhead.

In (5.2), and are constant once a specific RTOS and the simulator are given. Therefore, if CPU idle ratio can be obtained before simulation, designers can easily estimate the performance

gain of the proposed approach against the case when RTOS is actually executed on the ISS.

6. EXPERIMENTS

In this section, we show some preliminary experimental results on the performance improvement and on the accuracy. We consider a real-life multimedia application, a video phone, which consists of four tasks; an H.263 encoder, an H.263 decoder, a G.723 encoder and a G.723 decoder. They are assumed to be mapped to an ARM720T processor with 8KB unified cache. The RTOSes we used in the experiments are eCOS[9] and uCOS-ii[10]. We used ARMulator[8] for the processor simulator that runs on the dual Xeon 1.8GHz machine.

Depending on which RTOS is used and whether cache is used or not, different sets of task deadlines are used as shown in Table 1. Also, the priorities of the tasks are assigned as shown in Table 1. We assume that the period of a task equals to its deadline. The tick size is set to 10 ms for both eCOS and uCOS.

Even though the OS modeling technique is proposed for hardware/software cosimulation, we used the software simulator only for experiments to focus on the performance improvement due to the OS modeling. Thus we could ignore the effect of the hardware simulator performance on the experiments.

Table 1. Timing constraints of the example task group; 1tick is 368,640 cycles (=10ms)

Deadline	G.723 dec	G.723 enc	H.263 dec	H.263 enc
eCOS, cache ON	30 ticks	30 ticks	300 ticks	300 ticks
uCOS, cache ON	20 ticks	20 ticks	200 ticks	200 ticks
eCOS, cache OFF	400 ticks	400 ticks	4000 ticks	4000 ticks
uCOS, cache OFF	300 ticks	300 ticks	3000 ticks	3000 ticks
Priority	1	2	3	4

6.1 Performance Result

In the performance experiments, we classified the performance improvement into two factors. The first one is the improvement due to OS modeling itself (apart from synchronization overheads) and the second one is due to the removal of the time synchronization overheads.

Table 2 shows the performance improvement due to OS modeling itself. Simulation was performed with cache enabled and with the corresponding timing constraints defined in Table 1. For eCOS simulation, using the equation (5.2), the expected gain is 19.7% while the measured improvement in Table 2 is 18.2%. For uCOS simulation, we can obtain the expected gain similarly. It is 37.4% and the measured value is 38.0%. Recall that is the degradation ratio, the idle ratio and the RTOS ratio to the total simulation time.

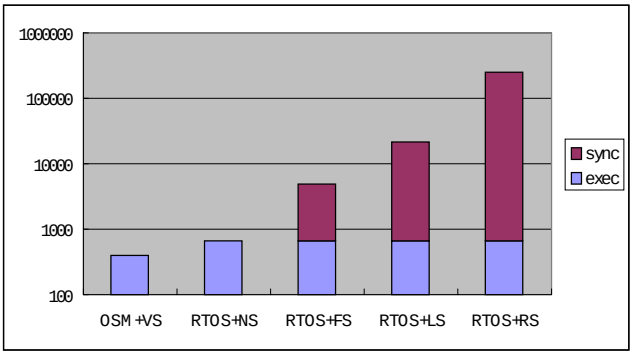

Figure 5. The time consumed to simulate uCOS-ii with cache enabled. The processor utilization is 73.0%

Table 2. Performance improvement due to OS modeling

	Estimated				Measured		
	s (%)	i (%)	r (%)	$1-\frac{ST_{model}}{ST_{RTOS}}$	ST_{RTOS} (sec)	ST_{model} (sec)	$1-\frac{ST_{model}}{ST_{RTOS}}$
eCOS	15	4.2	3.4	19.7%	905	740	18.2%
uCOS	15	27.0	1.0	37.4%	668	414	38.0%

Figure 5 illustrates the gain we can obtain from the removal of time synchronization. The first bar shows the simulation time from the proposed virtual synchronization approach with OS modeler. The time synchronization takes less than 1 second. It confirms that the virtual synchronization technique has nearly zero overhead of time synchronization. The second one shows the case when we simulate the tasks with RTOS without any synchronization. The third one is the case when we run RTOS with conservative approach, where time synchronization is performed every cycle with function call overhead of approximately 5.0 usec per simulated cycle. The fourth and fifth ones are the cases when we perform synchronization through TCP/IP socket call locally and remotely respectively. The measured the socket call overheads are 27.9 and 336.7 usec per simulated cycle respectively. Compared with the conservative approach where RTOS is run on the ISS, the proposed approach gives significant performance improvement due to the removal of time synchronization between component simulators

6.2 Accuracy Result

As the metric of time accuracy, we compute the error between the proposed RTOS modeling and RTOS simulation in terms of simulated times of task completion. Table 3 shows the results with eCOS and uCOS assuming that the cache memory is disabled. Since the RTOS overhead is accurately measured and almost constant at run time, the error is quite low. The execution cycles for the same task in two RTOSes are different because different compilers are used for each RTOS and these compilers use different library for floating-point arithmetic.

Since the proposed approach does not accurately model the cache affects, the error grows as demonstrated in Table 4. The error comes from the fact that the proposed approach underestimates the cold miss rate when a task starts or resumes as explained in section 4.

Table 3. Execution time error from the proposed approach for eCOS and uCOS-ii with cache disabled (time unit: cycles)

	Instances	eCOS	uCOS	exec. time (eCOS)	exec. Time (uCOS)
G.723 dec	100	0.06 %	-0.02 %	26,948,018	15,649,829
G.723 enc	100	-0.02 %	-0.05 %	31,101,618	17,629,837
H.263 dec	10	-0.03 %	-0.03 %	125,734,905	76,448,526
H.263 enc	10	-0.08 %	0.01 %	713,685,303	395,145,693

Table 4. Execution time error from the proposed approach for eCOS and uCOS-ii with cache enabled (time unit: cycles)

	Instances	eCOS	uCOS	exec. time (eCOS)	exec. Time (uCOS)
G.723 dec	100	-1.4 %	-7.3 %	1,765,217	921,301
G.723 enc	100	-1.5 %	-1.0 %	2,209,160	966,276
H.263 dec	10	-0.9 %	-1.3 %	11,610,644	6,707,270
H.263 enc	10	-0.6 %	-0.7 %	55,996,252	19,920,577

7. RELATED WORK

There has been no work on RTOS modeling for time accurate cosimulation with the ISS of a processor, to our best knowledge. On the other hand, there are some research results on the RTOS modeling for the transaction-level cosimulation with delay-annotated C-level software simulation. The RTOS modeling is addressed in Pia codesign environment[5]. In Pia, a software simulator blocks and synchronizes whenever it meets a receive operation. If an interrupt occurs, Pia preempts the current task execution and performs roll-back if necessary. Although interrupt handling is modeled with roll-back mechanism, preemptive scheduler modeling was not mentioned. No timing accuracy is reported in their work.

POLIS[3] also adapted RTOS scheduler modeling where a software task is synthesized to a timing annotated C code. A software task is executed atomically to produce the output results. If it is detected later that an interrupt or an input event for higher priority task arrives during task execution, the timing of the output events produced by the current task is delayed by the execution time of interrupt handler or higher priority task, which is similar to the proposed idea of time translation. They report that the accuracy was within 20% and this is due to the limit of host code execution with delay annotation approach. Cockx[4] also proposed a similar way of modeling of preemption. In his codesign environment, TIPSY, to reduce the context switch between software threads due to preemption, out-of-order execution of software threads is allowed. And timing adjustment is performed later for reconstruct the correct ordering of thread executions.

Yoo et al[6] proposed a fast timed simulation method by annotating delay in systemC simulation model. Instead of modeling the OS, they annotate the execution time to the OS code itself. The rationale is to make the simulation code and the synthesis code as much similar as possible. Since time adjustment is not used in this approach, synchronization should be performed either conservatively or optimistically.

8. CONCLUSIONS

In this paper, we extend the virtual synchronization technique with OS modeling for the case where multiple software tasks are executed under the supervision of a real-time operating system. The OS modeler models the RTOS overheads of context switching and tick interrupt handling as well as preemption behavior.

While maintaining the timing accuracy to an acceptable level below a few percents, we could reduce the simulation time drastically compared with existent conservative approaches by removing the need of time synchronization between simulators. It remains as a future work to model OS more accurately when cache is enabled

9. ACKNOWLEDGEMENTS

This work was supported by National Research Laboratory Program (number M1-0104-00-0015) and Brain Korea 21 Project. The ICT at Seoul National University provided the research facilities for this study.

REFERENCES

[1] Seamless CVE, http://www.mentorg.com/seamless/

[2] V. Zivojnovic and H. Meyr, “Compiled HW/SW co-Simulation”, *In Proc. 33rd DAC*., June 1996

[3] C. Passerone, “Fast hardware/software co-simulation for virtual prototyping and trade-off analysis”, *In Proc. 34th Design Automation Conf*., 1997

[4] J. Cockx, “Efficient modeling of preemption in a virtual prototype”, *In Proc. 11th RSP*, 2000, pp 14 –19

[5] K. Hines, “Pia: a framework for embedded system co-simulation with dynamic communication support”, *Technical Report UW-CSE-96-11-04*, 1997

[6] S. Yoo et al. “Automatic generation of fast timed simulation models for operating systems in SoC design”, *In Proc. Design Automation and Test in Europe*, Mar. 2002

[7] D. Kim et al. “Virtual synchronization for fast distributed Cosimulation of dataflow task graphs”, *In Proc. 15th International Symposium on System Synthesis*, Jun. 2002

[8] ARMulator, http://www.arm.comr

[9] eCOS, http://sources.redhat.com/ecos

[10] uCOS, http://www.ucos-ii.com

[11] K. Hagen and H. Meyer, “Timed and Untimed hardware software co-simulation: application and efficient implementation”, *In Proc. CODES*, Oct. 1993

[12] S. Yoo and K. Choi, “Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model”, *In Proc. 6th CODES*, Mar. 1998

[13] W. Sung and S. Ha, "Efficient and Flexible Cosimulation Environment for DSP Applications"*, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Issue on VLSI Design and CAD algorithms*, Vol.E81-A, No. 12, pp. 2605-2611, Dec. 1998

[14] CoCentric System Studio, http://www.synopsys.com /products/cocentric_studio

[15] CoWare N2C, http://www.coware.com/cowareN2C.html