Parallel Co-simulation Using Virtual Synchronization with Redundant Host Execution

Dohyung Kim†
Soonhoi Ha§
Rajesh Gupta†

†Department of Computer Science and Engineering
University of California, San Diego, USA
{dhkim, rgupta}@ucsd.edu

§School of Computer Science and Engineering
Seoul Nation University, Korea
sha@iris.snu.ac.kr

Abstract

In traditional parallel co-simulation approaches, the simulation speed is heavily limited by time synchronization overhead between simulators and idle time caused by data dependency. Recent work has shown that the time synchronization overhead can be reduced significantly by predicting the next synchronization points more effectively or by separating trace-driven architecture simulation from trace generation from component simulators. The latter is known as virtual synchronization technique. In this paper, we propose redundant host execution to minimize the simulation idle time caused by data dependency in simulation models. By combining virtual synchronization and redundant host execution techniques, we could make parallel execution of multiple simulators a viable solution for fast but cycle-accurate co-simulation. Experiments show about 40% performance gain over a technique which uses virtual synchronization only.

1. Introduction

Our focus is on efficient simulation of complete systems consisting of multiple concurrent components including multiple processors. Typically, the simulation of such systems consists of multiple simulators connected to each other reflecting component interactions. The efficiency of the overall system simulation is strongly affected by the extent of the inter-simulator interactions. Further, to reduce simulation time, we seek to build such simulation models for execution on a multi-processor system, here referred to as the “simulation host”.

In co-simulating multiple processors, each simulator should check events from all other simulators at every clock cycle. It guarantees that each simulator receives events from other simulators in the chronological order. Otherwise, causality errors may occur where a simulator receives a past event after it advances its local clock. To synchronize the advancement of local clocks of simulators, the conservative approach is to exchange control messages between simulators at every clock increment. Then this time synchronization overhead dominates the overall simulation time as the number of simulators increases or the simulator speed grows. For instance, if IPC overhead for unit message exchange between two simulators is larger than 10us, this limits simulation speed below 100K cycles/sec. Actual simulation performance is much worse.

Some recent approaches address this issue by reducing the synchronization points using software analysis technique and the virtual synchronization technique. In the former approach, the analysis can determine the time when each simulator should be synchronized [1]. In the latter approach, time synchronization can be avoided by separating trace-driven architecture simulation from trace generation from component simulators [2]. Traces from component simulators are captured only at the boundary of actual data exchanges in algorithm specification.

With the reduced synchronization overhead, the performance of parallel co-simulation is now limited by the parallelism of a simulated system. If one simulator waits for data from another simulator, such data dependency between simulators serializes executions of two simulators.

To overcome this limitation, we propose redundant host execution of algorithm specification. In this paper, we redundantly compute data required by a simulator in the simulation host. Then the data computed from the host execution are fed into the simulator even before data is delivered from the other simulator. This eliminates the simulation idle time that is caused by data dependencies between simulators in parallel execution of multiple simulators.

![Figure 1: Difference between traditional parallel co-simulation and proposed parallel co-simulation when a data dependency exists between simulators](image-url)
Figure 1 illustrates how the proposed approach can achieve efficient parallel co-simulation with data dependency. We assume that func_A is executed on the sim_1 simulator and func_B on the sim_2 simulator. Func_B is dependent on func_A’s output. In a traditional co-simulation approach, func_B on the sim_2 simulator can be executed only after the sim_1 finishes executing func_A and delivers the output to func_B running on the sim_2. However, in the proposed approach, the simulation host where the simulation is performed executes func_A redundantly and immediately produces data for func_B. Then we can initiate func_B on the sim_2 simulator by delivering the data from the redundant host execution before the completion of func_A on the sim_1.

3. Trace-driven co-simulation using virtual synchronization

Based on a computation model which defines algorithm behavior precisely, the virtual synchronization technique determines when each simulator should be synchronized with other simulators. Moreover, when it performs synchronization, it does not synchronize the local time of each simulator to the global time. Instead, each simulator delivers relative times between data samples. Then, as a centralized co-simulation controller, the simulation kernel transforms the relative times of data samples to the global times. In this way, the local time of each simulator is virtually synchronized to the global time in the simulation kernel.

As illustrated in Figure 2, a conventional synchronization scheme synchronizes all simulators to the global clock. In contrast, virtual synchronization takes the relative times \((t_1, t_2, t_3)\) from two simulators and transforms those times to the global times \((t_0+t_1, t_0+t_1+t_2, t_0+t_1+t_2+t_3)\) in the simulation kernel. Thus the roles of data sample generation and timing management are separated to the component simulators and the simulation kernel respectively.
For more accurate time management considering communication architectures and dynamic behaviors, component simulators generate access traces between actual data exchanges and the trace-driven architecture simulator is separated from the simulation kernel in trace-driven co-simulation using virtual synchronization [2]. In this scheme, the simulators store during execution all accesses to the architecture components (resources) which may cause conflicts with other simulators. We define an access to an architecture component as a resource access trace. Note that all resource access traces have relative times between traces to apply virtual synchronization. The simulation kernel delivers input data to a simulator, executes the simulator and acquires output data with resource access traces as shown in Figure 3. This is the first part of trace-driven co-simulation.

Once resource access traces are acquired, the second part of trace-driven co-simulation, called trace-driven architecture simulator, transforms the relative times in the resource access traces to the global times by considering conflicts on the architecture resources. The architecture simulator resolves conflicts on a processor by modeling operating system timing behavior and on a memory by modeling communication architecture. The trace-driven co-simulation is similar to Metropolis [9] and Artemis project [10] except that how and which traces are obtained. They use the execution of algorithm specification to drive the simulation of architecture specification while we obtain all resource access traces from component simulators.

Note that the simulation kernel plays the role of broker between component simulators and the trace-driven architecture simulator. After the simulation kernel acquires resource access traces from one simulator, it directly provides the traces to the architecture simulator and starts the simulator. If the architecture simulator consumes all resource access traces or cannot advance the global time safely, it requests new traces from the simulation kernel.

4. Parallel co-simulation techniques

Section 4.1 explains a parallel scheduling algorithm for trace-driven co-simulation and its limitation. Section 4.2 introduces redundant host execution technique and section 4.3 the device model for host execution.

4.1 Parallel scheduling of multiple simulators

In the trace-driven co-simulation, the simulation kernel, as a central controller, repeats the following tasks iteratively: (1) It determines a simulator to execute, (2) invokes the simulator with input data, (3) waits for output data with access traces, and (4) evaluates them using the architecture simulator. Figure 4 shows some iterations of such sequence, where the number in the simulation kernel chart indicates which step it currently performs. In the figure, the example shown in Figure 3 is performed by the trace-driven co-simulation of [2]; Three simulators are sequentially executed as illustrated in Figure 4 because the simulation kernel waits until an execution of a simulator finishes in each iteration.
pendent of each other. At each iteration, simulation kernel invokes at most one simulator. Therefore, to make simula-

tors be overlapped across iterations, the simulation kernel

may not be blocked on step (3) unless the simulator is busy processing data sent earlier. Figure 5 illustrates how
the proposed scheme works as follows.

In Figure 5, the simulation kernel invokes simulator 1 at
the first iteration but is not blocked on step (3) since simu-
lar 1 is not processing any data at that time. So it goes to
the next iteration to invoke simulator 2. Again it is not
blocked on step (3) since simulator 2 has been idle. At the
third iteration the simulation kernel examines simulator 1
again and finds out that the simulator is processing data
that were sent earlier (at the first iteration). So it is
blocked on step (3) until it received resource access traces
from simulator 1. Note that the simulation kernel does not
examine simulator 3 since it knows that function C is not
executable yet due to data dependency. Simulator 3 is
invoked at the fifth iteration after func A and func B fin-
ishes their executions.

**Equation 1. The simulation time for one iteration**

\[ \text{iter}_i = \max (\text{iter}_i, \text{sim}_k - \sum_{i=1}^{k-1} \text{iter}_i) \]  

Since the waiting time in step (3) is usually much larger
than the other terms, the performance of this basic parallel
co-simulation is bound to parallelism of a simulated algo-

rithm. In Figure 5, simulator 3 waits for data from both
simulators 1 and 2 since the simulated system of Figure 3
has such dependency. This limitation is overcome by re-
dundant host execution.

### 4.2. Redundant host execution technique

In the proposed technique, we redundantly execute al-
gorithm specification on the simulation host. Then, we
provide the output data from the host execution to a simu-
lator before data from other simulators are available. Note
that this technique is possible since global time manage-
ment in trace-driven architecture simulator is separated
from trace generation from component simulators. Thus,
we can reconstruct the out-of-order execution of the simu-
lation model by the redundant host execution. It comple-
mentarily accelerates trace generation from the simulators
in the trace-driven co-simulation.

We add another step for the redundant host execution in
an iteration of the simulation kernel: in step (5) it executes
the algorithm specification at the host machine before step
(2) as shown in Figure 6. Usually the host execution
(>1GIPS) is much faster than the most advanced proces-
sor simulator (<1MIPS). Thus, the output is available in-
stantly. Then when it determines the next simulator to
invoke for the next iteration, it has more chances to exe-
cute other simulators concurrently because data can be
provided by the redundant host execution. In Figure 6,
simulator 3 receives data from host executions of func A
and func B that are performed at the first and the second
iteration respectively. Thus all three simulators run in par-
allel. As can be observed from Figure 6, redundant host
executions tend to reduce the waiting time of step (3) so
that the overhead of host execution is negligible in most
cases. Comparing to Figure 4, we do not need to receive
output data from simulators because the redundant host
execution already produces them.
with the dominant simulator. The simulation time for that iteration represents the second term of equation (1). The simulation time of all other iterations will take the first term. If we add the simulation times of all iterations, we obtain the total simulation time that becomes \( \sum \text{sim}_k \) plus a few terms associated with initial iteration steps before the first invocation of the dominant simulator. In short, if there is one dominant simulator, the total simulation time will be bound to the simulation time consumed for the dominant simulator. Figure 7 illustrates this fact graphically where the simulation times of most iterations are covered by the simulation time consumed by simulator 1, the dominant simulator.

4.3 Device model using host execution

To enable redundant host execution, we need to capture two types of data. One is input data that would have been generated by other simulators but are now generated by the host execution. Input data for a simulator should be captured before a host execution because the host execution may manipulate the input data.

The other is device data which are accessed through system calls during the host execution. Device data would have been provided by device modeling in processor simulators [11][12]. To provide device data during host execution, however, we override system calls by using “#define” macro in algorithm specification at the host machine and simulators respectively as shown in Figure 8.

First, overridden system calls at the host machine stores return values and data from the devices during host execution. Second, the simulation kernel delivers input data and device data together when it invokes a simulator. Finally, device data are delivered to the overridden system calls at the simulator. Because calling sequences of system calls at the host execution and the simulation execution are identical throughout the simulation, we use FIFO queues without any identification for system calls when we deliver device data.

For example, to read a file from HDD, we replace `read` and `lseek` system calls with `read_device` and `lseek_device` function calls. `read_device` function stores the size and the data from `read` system call. `lseek_device` function stores the return value from `lseek` system call. Then when invoking the simulator at the third step, it delivers the input data and device data together.

In the simulator, overridden `read_device` and `lseek_device` functions store arguments into specific addresses. Then they put a unique value at the special control address, which gets caught by `mem_write` function at the simulator interface. In `mem_write` function, it reads device data from the host execution and copies them to the application area.

5. Experiments

Figure 9 shows a DIVX player example which is composed of three tasks: an H.263 decoder, an MP3 decoder and an AVI file reader. While each task has multiple function blocks, we only show the internal blocks of the H.263 decoder task. The H.263 decoder task consumes 95% clock cycles on the simulator in sequential execution. It is composed of header decoder, dequantization (DQ), inverse discrete cosine transform (IDCT), motion compensation (MC) and display blocks.

![Simplified view of DIVX player example](image)

As shown in Table I, we use two different mappings using 2 processors and 5 processors respectively. We implemented the proposed technique in PeaCE framework [13]. For processor simulator, we use ADS 1.2 from ARM and the simulation is executed on Linux 2.4 with Xeon 2.6Ghz dual CPUs. Each experiment decodes 11 frames, which takes 34.3 seconds and requires 55M cycles if simulated in one simulator.

<table>
<thead>
<tr>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
<th>Proc 4</th>
<th>Proc 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>IDCT, MC</td>
<td>Others</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>IDCT, MC</td>
<td>MP3, Display</td>
<td>Others</td>
<td></td>
</tr>
</tbody>
</table>

For each architecture, we experiment three different co-simulation schemes: original trace-driven co-simulation using virtual synchronization technique (original), parallel...
cosimulation without host execution (parallel), and parallel cosimulation with redundant host execution (proposed).

In the experiments, the host execution of the entire algorithm only takes 0.6 second and the simulation time except for the waiting time of step (3) is less than 2 seconds in all cases. Therefore the total simulation time is mostly taken by simulator execution time without time synchronization overhead.

The experiments show that the proposed approach reduces simulation time using parallel simulation by 40% and 45% for two cases respectively compared with the previous approach [2] that already showed 43 times better performance than the traditional conservative approach.

Table II. Performance comparison between original virtual synchronization technique and the proposed approach with/without the host execution

<table>
<thead>
<tr>
<th>Simulator #</th>
<th>Original</th>
<th>Parallel</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>67.8s</td>
<td>54.6s (19%)</td>
<td>40.6s (40%)</td>
</tr>
<tr>
<td>5</td>
<td>66.8s</td>
<td>51.4s (23%)</td>
<td>36.7s (45%)</td>
</tr>
</tbody>
</table>

Figure 10 shows how the waiting times for simulators are reduced from the proposed redundant host execution technique. The total simulation times of the proposed approach are dominated by the simulation times of some dominant simulators. In these experiments, there was no single dominant simulator.

![Figure 10: Comparison of waiting times between the original cases and the proposed cases](image)

6. Conclusion

This paper presents how to parallelize execution of processor simulators based on virtual synchronization technique. Moreover, to reduce data dependent latency between simulators, we propose the redundant host execution technique to compute data required for the simulators early. However, the proposed technique is only applicable for time invariant function as virtual synchronization does. We have implemented the technique in PeaCE framework. Results demonstrate the usefulness of the proposed technique in multi-processor co-simulation environments. Experiments using a DIVX player example show about 40% and 45% reduction in simulation time using two and five processor simulators respectively, compared with our previous virtual synchronization technique that is already 43 times better than the traditional conservative co-simulation. As future work, we will extend the proposed technique for time variant function.

Acknowledgement

Our work was partially supported by a gift from Intel Corporation, a grant from UC Discovery program and a post doctoral research program from Ministry of Information and Communication (MIC, ROK). We are grateful to the anonymous reviewers for their insightful comments.

References