Center for Embedded Computer Systems
University of California, Irvine

A Distributed Parallel Simulator for Transaction Level Models with Relaxed Timing

Weiwei Chen, Rainer Dömer

Technical Report CECS-11-02
May 31, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{weiweic, doemer}@uci.edu
http://www.cecs.uci.edu/

A Distributed Parallel Simulator for Transaction Level Models with Relaxed Timing

Weiwei Chen, Rainer Dömer

Technical Report CECS-11-02
May 31, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{weiweic, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

Embedded systems are captured and refined into transaction level models (TLM) written in System Level Description Languages (SLDLs) through the top-down synthesis design flow. However, the traditional single-thread discrete event (DE) simulator cannot use the explicit parallelism in these models for effective simulation. In this paper, we present an efficient scalable distributed parallel DE simulation system with relaxed timing synchronizations. Our distributed simulation engine is suitable for un-timed or approximate-timed TLMs. We demonstrate the benefits of the distributed simulator using a basic pipeline model and a case study on a JPEG encoder application.

Contents

List of Figures

List of Tables

A Distributed Parallel Simulator for Transaction Level Models with Relaxed Timing

Weiwei Chen, Rainer Dömer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{weiweic, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

Embedded systems are captured and refined into transaction level models (TLM) written in System Level Description Languages (SLDLs) through the top-down synthesis design flow. However, the traditional single-thread discrete event (DE) simulator cannot use the explicit parallelism in these models for effective simulation. In this paper, we present an efficient scalable distributed parallel DE simulation system with relaxed timing synchronizations. Our distributed simulation engine is suitable for un-timed or approximate-timed TLMs. We demonstrate the benefits of the distributed simulator using a basic pipeline model and a case study on a JPEG encoder application.

1 Introduction

Multiple processing elements are integrating onto one chip (MPSoC) for modern embedded computer systems to provide various functionalities and meet tight design constrains, e.g. real-time computation, small chip size, and low power consumption. The processing elements, including general-purpose CPUs, application-specific instruction-set processors (ASIPs), digital signal processors (DSPs), as well as dedicated hardware accelerators and intellectual property (IP) components, can respectively accomplish complicated computing or controlling tasks, and are connected as a network in the system for communications. The complexity of the system poses great challenges to design and validation.

Research works have been done for efficient system design and validation in different areas, including system level description languages, systematic design methodologies, higher abstract level

modeling, as well as advanced parallel simulation engines. Modern C-based System-level Description Languages (SLDLs), like SpecC [10] and SystemC [11] are popular for describing both the hardware and software of embedded systems, and are supported by tools like System-on-Chip Environment [7] and Forte Design [8] for design space exploration and model synthesis. A systematic top-down design methodology, called Electronic System Level (ESL) design, allows the designers to capture their systems at high abstraction level by SLDLs and refine them step by step down to detailed implementations. While the refinement goes down to lower levels of abstraction, the models contain more details and become more complicated which brings obstacles to efficient validation, e.g. fast simulation and debugging. Transaction-level Modeling [11] is another well-accepted approach to efficient model validation which abstracts away the low level implementation details and separate communication and functional units. The high simulation speed is traded in for low timing accuracy by using TLM.

In this paper, we are focusing on the parallelization of SLDL simulation engines for efficient model simulation. A new distributed parallel SLDL simulator is proposed for the consideration of using the computation resources of multiple geographically distributed machines connected with network. Relaxed time synchronization is applied for fast simulation speed but trade in timing accuracy for simulation speed like Transaction-level Modeling.

After a brief review of existing efforts on fast SLDL simulation in Section 2, the new distributed parallel simulator is proposed in Section 3. The design of the inter-host channel is discussed for inter-host communication and synchronization. In Section 4, we show the simulation results of a JPEG encoder design at different abstraction levels. The speedup scaled to the number of simulation hosts is achieved and the timing inaccuracy is very subtle. Conclusions are made and the future works are discussed finally in Section 5.

2 Related Work

Discrete Event (DE) simulation is used for both SpecC and SystemC SLDLs. DE simulation is driven by events and simulation time advances. However, the traditional single-thread DE simulation kernel is an obstacle to utilize any potential parallel computation resources for performance progresses [12].

There has been considerable effort on parallelizing DE simulation. A well-studied solution is Parallel Discrete Event Simulation (PDES) [2, 9, 14].

Multi-core parallel simulation is discussed in [3] and [15]. The simulator kernel is modified to issue and properly synchronize multiple OS kernel threads in each scheduling step. This allows parallel execution of the models described in SLDLs on multi-core machines. However, synchronization protection overhead is introduced for safe communication simulation. The gain of using multi-core simulator can be very limited when the model simulated has tight timing constraints which reduces the possibility of simulation parallelism.

Distributed parallel simulation, on the other hand, partitions the model and deploys each piece onto geographically separated machines which talk with each other via network, and perform their own work concurrently. [16] deals with the distributed simulation of RTL like SystemC models without experiment results. Clusters with single-core nodes are targeted in [4] which uses multi-

ple schedulers on different processing nodes and defines a master node for time synchronization. A distribution technique for an arbitrary number of SystemC simulations is proposed in [12] for distribute functional and approximate-timed TLMs. [6] presents a distributed SystemC simulation environment executed on a cluster of workstations using message-passing library. However, the central global synchronization applied in [4, 12] and [6] can be the bottle neck of the simulation system so that a great part of the parallelism potential can be wasted. Decentralized simulation time synchronization is then proposed in [5] to reduce the central synchronization overheads. Each cluster node goes on running its own delta cycle loop, and only communicates with the others at the end of the loop for message passing or time advancement.

Our proposed fast SLDL simulator is on the track of distributed parallel simulation. The idea is to achieve simulation speedup by separating workloads onto multiple distributed machines and relaxing the time synchronization between them. Moreover, the simulator on each host does not have to take care of the simulation status of all the other hosts in the system, but only communicates with the hosts and synchronizing the timing information through our inter-host channels if connected. Inter-host channels are only used when two hosts respectively contain functional units who have to communicate across the hosts after partitioning.

3 Distributed Parallel Discrete Event Simulation

MPSoC models written in SLDLs usually contain explicit parallelism which makes it straightforward to increase simulation performance by partitioning the model and executing each part concurrently on different hosts. Even though each part separated on different hosts may be loosely coupled, communication cannot be ignored for correct functionalities. Furthermore, timing accuracy will lose when the model sub-pieces are simulated on separate simulators which have their own local time counters. Thus, cares must be taken for proper communication and timing synchronization between the distributed simulators.

Our distributed parallel simulation system in this paper is based on our previous work for multi-core parallel SLDL simulator [3] to achieve parallelism as much as possible. The communication and synchronization idea can also be applied on traditional single-thread SLDL simulators. Without loss of generality, we assume the use of SpecC SLDL here (i.e. our technique is equally applicable to SystemC).

3.1 Formal Definitions

To formally describe the distributed parallel DE simulator, we define the following data structures and operations:

1. Definition of the distributed DE simulator (DISTSIM):
   DISTSIM = (**H**, **C**, **M**), where
   - **H**=$\{h \mid h$ is the host in the simulation system.$\}$,
   - **C**=$\{ch \mid ch$ is the inter-host communication channels.$\}$,
   - **M**=$\{m \mid m$ is the inter-host messages.$\}$

2. Definition of the simulation hosts ($\forall h \in$ **H**):
   Each simulation host runs its own DE simulator. The local data structures and operations are the same as those defined in [3], section III-A. Briefly, they are

   - Thread queues in the simulator: **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**, **COMPLETE**}.
   - Operations on thread th: **Go**(th), **Stop**(th), **Switch**(th).
   - Operations on thread with set manipulations: **Create**(), **Delete**(), **PICK**(), **MOVE**().
   - Simulation invariants and initial state of the simulator.

   We focus on the essential definitions for describing the hosts in the distributed simulation system below. The others still remain the same in the local simulator.
   $\forall h \in$ **H**, h = (**QUEUES'**, $curr\_t$, $curr\_delta$, **E**), where

   - $curr\_t$ is the local time counter,
   - $curr\_delta$ is the local delta cycle counter,
   - **E** is the local event list, and
   - **QUEUE'** = {**READY**, **RUN**, **WAIT**, **WAITFOR'**, **COMPLETE**}. Here, **WAITFOR'** is a super set of **WAITFOR** with waitfor threads and messages.

3. Definition of the inter-host communicate channels ($\forall ch \in$ **C**):
   Inter-host communication channels used in our distributed simulator are point-to point FIFO channels. They share information between two hosts via the network. Each channel maps to one TCP/IP socket.
   $\forall ch \in$ **C**, ch = ($socket$, $size$, ch_type, $portNo$, $buffer$), where

   - $socket$: is TCP/IP socket used for inter-host communication via network,
   - $size$: is the size of the FIFO buffer,
   - ch_type: identifies whether the current host h is the sender or the receiver of the this channel,
   - $portNo$: is the port number of the corresponding socket,
   - $buffer$: is the storage of the data in the channel.

   The details of the inter-host communication channel will be discussed in Section 3.4.

4. Definition of the inter-host messages ($\forall m \in$ **M**):
   Messages are passing via network between hosts in the distributed simulation system.
   $\forall m \in$ **M**, m = (m_type, $timestamp$, $content$, th_recv, ch), where

   - m_type: is the type of this message (data or event),
   - $timestamp$: is the timestamp of the message (local current time of the sender),
   - $content$: is the content of this message (data content or event information),

- *th_recv*: is the data receiver thread,
- *ch*: is the corresponding inter-host channel who delivers this message.

5. Definition of the extended **WAITFOR'** queue for each simulation host:
WAITFOR' = $\{w \mid w$ is a thread $th \in$ **WAITFOR**, or a message m, $m.timestamp > curr\_t$ when m is received. $\}$.

3.2 Distributed Discrete Event Simulation System

The distributed DE simulation system is a tuple (**H**, **C**, **M**) as defined in Section 3.1. Figure 1 shows a brief picture of how the distributed DE simulator works. It consists of multiple simulation **hosts** running local DE simulator respectively. Each simulation host has several working threads and its own scheduler. Communication and synchronization is done via **messages** passed through inter-host communication **channels**.

In order to receive cross-host messages on time and handle them properly, a listener thread *listener* is introduced. *listener* is created at the beginning of the local simulation program and keep alive as long as the simulation program is working. Only *listener* will receive the messages from outside through the TCP/IP sockets. It will then pass them to either the working *receiver* thread or the scheduler according to the timestamp of the messages. The *scheduler* helps to deliver the message with future timestamp later at a certain scheduling step. Relaxed timing synchronization is done at the scheduling step when the messages with timestamps are consumed.

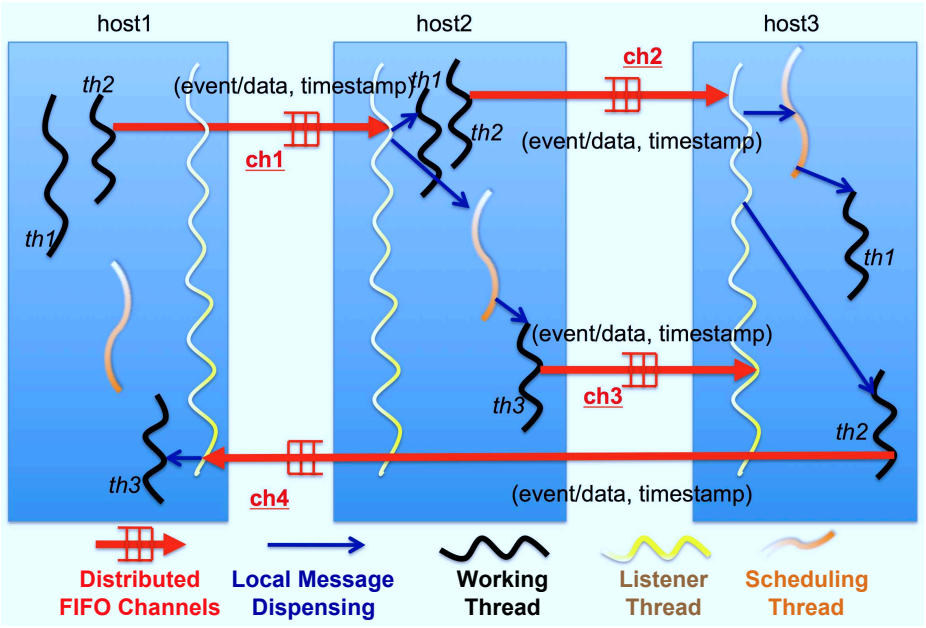


Figure 1: The architecture of the distributed DE simulator

3.3 Relaxed Timing Synchronization and Local DE Simulation Scheduling

To relax the timing synchronization for better simulation performance, our distributed simulation engine does not have a global monitor or the shared time counter. The host does not have to synchronize with the others at each delta cycle and time cycle, or behave as a slave who waits for the master's command for proceeding. Simulation hosts only communicate via inter-host channels. Local simulation time is adjusted according to the timestamps of the messages from the other connected host for synchronization.

The motivation for this timing synchronization mechanism is that two simulation hosts can do their own simulation work at different speed if they do not actually communicate with each other. This releases the hosts from central synchronizations at each scheduling steps. However, for a timed model, the simulation time is implicitly shared for all the functional behaviors in the model. In other words, all the functional units communicate with each other through the time counter. The time counter distribution onto different simulation hosts can introduce the loss of timing accuracy. But, as the intrinsic idea of TLM modeling, the precision of the timing information can be traded in a little bit for simulation speed. Or in other words, it is harmless to sacrifice some of the timing accuracy of an approximate-timed model, like TLM, to gain simulation progresses.

The timing synchronization is done through inter-host message passing on each simulation hosts. Messages from outside carry the timestamps from the source hosts. These timestamps help the current host to know the progress of its partner in the system, and provide guidance of timing adjustment to catch up the pace of the others. Messages are handled by both the *listener* thread and the *scheduler* before they are used by the receiver/sender of the inter-host channels. Here, *'use'* means use the messages (store the data in the buffer of the FIFO channel or notify the event), and *'handle'* means receive the messages from the socket and put them in a specific order for consumption.

In this section, we discuss the mechanism of message handling. The idea is to use the message as soon as possible when its timestamp is earlier or equal than the local time counter. Otherwise, the *listener* will put the message with *future* timestamp into **WAITFOR'** queue in the order of the timestamp and wait for the scheduler to deliver it when the specific time comes.

Figure 2 shows the control flow of the listener thread who receives the messages from the network sockets. L is the central lock for scheduling resource protection, and *simstate* is a flag for the status of the simulator on current host. *simstate* can be *BUSY* stating that the local simulation is active, or *IDLE* meaning the simulation is paused and wait for external triggers. The *listener* thread just keep listening to the network sockets for any incoming messages. When one messages is detected, the listener handle the message accordingly. The scheduler will be called then if local simulation is currently paused since the message is the external trigger. The central lock L is used for safe usage of the scheduling resources.

Figure 3 shows the control flow of the local simulation scheduler. Compared with non-distributed simulation scheduler in [3], the part for time cycle advancement is extended. Either a thread or a message will be picked up from the **WAITFOR'** queue in the time cycle handling part. If the one with the earliest future timestamp is a thread, the scheduler behaves in the same way as the non-distributed one. Otherwise, the scheduler will *use* the message (put the data into the channel buffer, or notify the event), check whether any thread in the **WAIT** gets its waited event notified then, and continue with the following steps. Local time *curr_t* will be advanced to the specific earliest

```
1  while(1)
   {
3    select from the incoming channels;
     receive the message m from chnl.socket;
5    Lock(L);
     if(m.timestamp <= curr_t){
7      if(m is data){
         chnl.listenerSend(m);
9      }
       else{ // m is event
11       notify m;
       }
13   }
     else{
15     put m into WAITFOR';
     }
17   if(simstate == IDLE){
       schedule();
19   }
     unLock(L);
21 }
```

Figure 2: The control flow of the listener thread

future timestamp. In distributed simulation, the simulator does not stop when the **READY** queue is empty after time cycle handling. The simulator will stay idling since none of the local threads are active, but still be alive to wait for external data or events. Thus, deadlock cannot be detected by checking whether the **READY** queue is empty after time advancement tries. Due to limited paper spaces, we leave the deadlock detection out of the scope of this paper.

3.4 Inter-host Communication Channels

The inter-host communication channel helps for data exchanges and timing synchronization among the distributed simulation hosts. As defined in Section 3.1, the inter-host communication channels are FIFO channels connecting two simulation hosts. Both sending and receiving operations behave in blocking fashion. Moreover, there is none additional memory units in between two hosts. Cares must be taken for the storage location of the internal data structure and proper way of communication operations.

The *sender* and the *receiver* of the inter-host channel are two threads on two different hosts. The

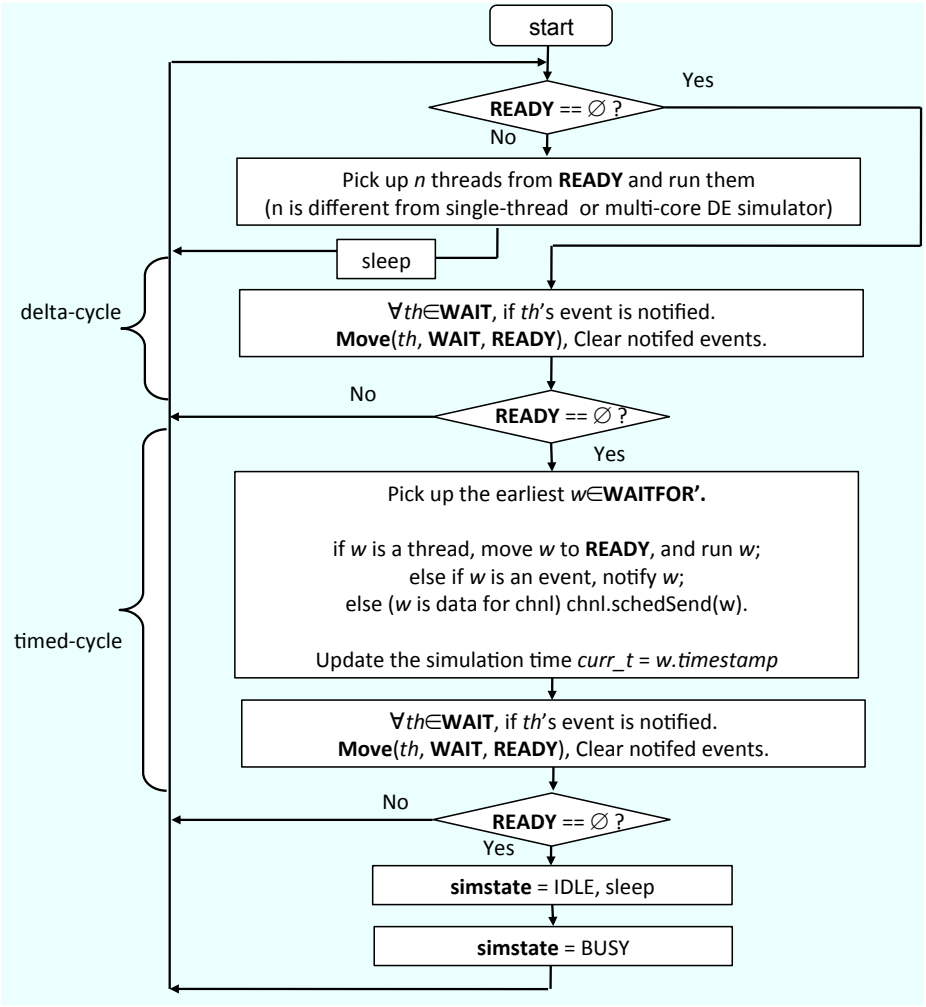


Figure 3: The local DE scheduler of the distributed host

hosts are ordered in the system. We always assign the host with lower order number as the server of the socket, and the host with higher order number as the client. The internal data structures, e.g. the storage buffer, synchronization flags (*wr*, *ws*) are stored on the *receiver* host. Both the *sender* and *receiver* host have the instance of the inter-host channel while the *ch_type* of the channel are different. The internal events, e.g. *eSend*, *eRecv* have their own copies on both hosts. However, only one of the copies will be used, e.g. *eSend* used by the sender host, *eRecv* used by the receiver host.

Five operations are defined for inter-host channel communication operations:

- **send**(*d*): send data *d*,
- **receive**(*d*): receive data *d*,

- **listenerSend**(d): send eternal data d locally by the listener,
- **schedSend**(d): send eternal data d locally by the scheduler,
- **distNotify**(e): notify the event e to the remote sender host via network.

The *sender* host sends the data d via the network socket to the receiver. The *listener* thread on the *receiver* host will catch the data and tell the *sender* whether to block or not according to the buffer space availability and timestamp comparison. If the feedback is blocked, the *sender* will wait on the *eSend* event, and release the CPU for other threads; otherwise, the *sender* continues. The blocked *sender* thread will be notified to continue when the last data sent is consumed by its remote *receiver* by event notification via network.

On the receiver host side, both the *listener* and *scheduler* behave as the local "sender" of the inter-host channel who do the actual data storing into the channel $buffer$. There are slight differences between these local sendings. For the *listener*, it stores the data into the buffer, sends feedback to the *sender* thread, manipulates the synchronization flags, and issues event notification to the local *receiver* thread if necessary. For the *scheduler*, it stores the data, manipulates the synchronization flags, issues event notifications to the local *receiver* thread as well as the remote *sender* thread if necessary .

Figure 4 shows the algorithm of the inter-host channel communication operations.

3.5 Model Partitioning

The rules for model partitioning onto different simulation hosts by using our simulation engine are as follows:

1. The functional behavior units simulated on different hosts cannot use shared variables, except for the simulation time counter.
2. Inter-host communication is done by point-to-point FIFO channels.

In this paper, we manually partition the models by duplicating copies for each host, removing unused behavior units from each copy, and replacing the intra-host channel with proper configured inter-host channel when necessary. General partitioning rules e.g. balanced workloads on different simulation hosts lead to better simulation performance, and behaviors between which communications are frequent shall not be separated, also applicable to the model partitioning when using our distributed simulation engine.

4 Experiments and Results

To demonstrate the improved simulation speed of our distributed DE simulator, we show two sets of experiment in this section.

```
send(d)
{
  Lock(L);
  socket.send(data d);
  socket.recv(feedback);
  if(feedback == blocked){
    wait eSend;
  unLock(L);
}


schedSend(d)
{ // L is locked;
  buffer.store(d);
  if(n < size){
    distNotify(eSend);
  }
  else{
    ws ++;
  }
  if(wr){
    notify eRecv;
  }
}

distNotify(e)
{
        socket.send(event e);
}

// wait() and notify() here do not acquire the central Lock L
// which is already acquire outside the functions.
```

```
receive(d)
{
  Lock(L);
  while(!n){
    wr ++;
    wait eRecv;
    wr --;
  }
  buffer.load(d);
  if(ws){
    distNotify(eSend);
  }
  unLock(L);
 }

 listenerSend(d)
 { // L is locked;
   buffer.store(d);
   if(n < size){
     socket.send(not blocked);
   }
   else{
     socket.send(blocked);
     ws ++;
   }
   if(wr){
     notify eRecv;
     if(simstate == IDLE){
       schedule();
     }
    }
   }
```

Figure 4: The algorithm of the inter-host channel communication operations

4.1 A basic pipeline without timing synchronization

As shown in Figure 5, our basic pipeline model contains N parallel stages with input and output ports connected by FIFO channels. Each stage 1) performs n_{flop} dummy floating point multiplications to emulate the workload in each execution iteration, and 2) waits for data from previous stage to start one iteration and passes data as the trigger of next stage's execution.

The model is parametrizable and un-timed. We use it to show 1) the effect of computation/communication ratio on simulation performance, 2) the scalability and the promising speedup of the distributed DE simulator.

We simulate this pipeline example 100 iterations for each stage with different number of total stages (N = 6, 12, 18, 24), and different scale of computation loads (n_{flop} = L, $LlogL$, $L*L$, where L = 1024). The model is partitioned in two ways: 1) simulation on 2 hosts, each with $N/2$ stages,

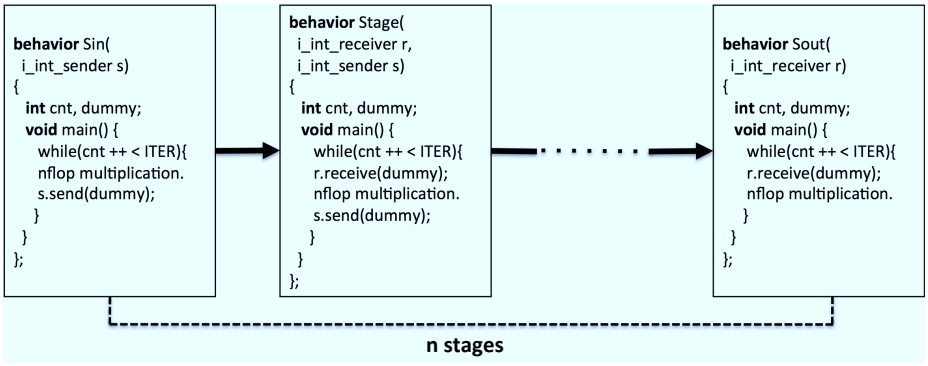
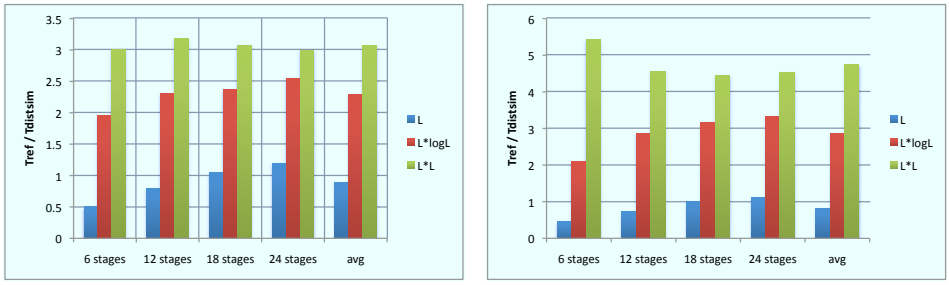


Figure 5: The pipeline example

and 2) simulation on 3 hosts, each with $N/3$ stages. Figure 6 shows the experiment results by using the host PCs with Intel (R) Core(TM) 2 Duo CPU E6550 at 2.33 GHz. The reference is the model simulation time by using the traditional single-thread DE simulator. The simulation speedup is shown by the ratio of the simulation time of the reference single-thread simulator versus the simulation time of our distributed simulator ($T_{ref}/T_{distsim}$). When computation load is low ($n_{flop} = L$), the speedup of distributed simulation versus the reference one is less than 1 due to the delay of using network sockets and synchronization protections. However, these overhead can be ignored when the computation/communication ratio is high ($n_{flop} = LlogL$, $L * L$). The average speedup is **3.07** for 2 hosts and **4.73** for 3 hosts when $n_{flop} = LlogL$. This speedup is greater than the theoretic speedup of both single-thread simulator (speedup = 1) and multi-core simulator (speedup = number of cores in the PC, which is 2 here) on one host.



(a) Results for simulation on 2 hosts

(b) Result for simulation on 3 hosts

Figure 6: Experiment results for the synthetic pipeline example

4.2 Real case study: TLMs of a JPEG Encoder

The JPEG Encoder is a real-world application [13], [1]. We use the refinement-based framework for heterogeneous MPSoC design, System-on-Chip Environment (SCE) [7] to perform the top-down synthesis design flow. Figure 7 shows the diagram of the JPEG encoder example.

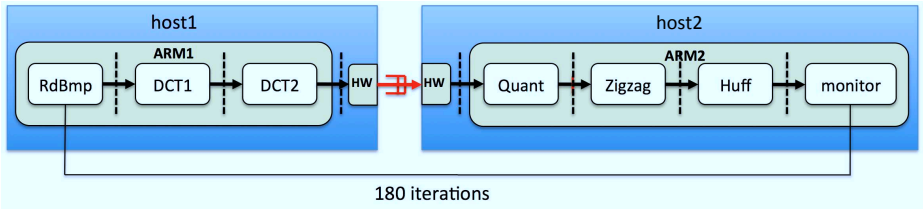


Figure 7: The JPEG Encoder example

For distributed simulation model partitioning, we insert a transceiver model between *DCT*2 and *Quant* and encapsulate the inter-host communication channel in this transceiver for the TLM models generated at different abstraction levels. The architecture mapping for this design is as follows: two ARM7TDMI processors at 100MHz is allocated, one for the first three modules (*RdBmp*, *DCT*1, *DCT*2), and one for the last four modules (*Quant*, *Zigzag*, $Huff$, *monitor*); the transceiver in between is mapped to a standard hardware unit. We choose priority-based scheduling for the tasks in the processor and allocate two AMBA AHB buses for communication between the processor and the hardware units.

We generate the models at different abstraction levels of the JPEG encoder by using the SCE refinement tool, including specification (*spec*), architecture (*arch*), scheduling (*sched*), network (*net*), communication (*comm*) as well as the implementation models. These models are timed and described in SpecC SLDL. The first four models (spec, arch, sched, net) at the higher abstraction levels have fewer timing details and take very short time for simulation. Due to the overhead of network communication and synchronization approximation, we demonstrate the efficiency of our DE simulator by using the lower abstract level models with more timing and implementation details. The models we simulate are: the pin-accurate model (**commPAM**) that refines the individual point-to-point links down to an implementation over the actual communication protocol and the corresponding transaction-level model (**commTLM**) that abstracts away the pin-level details of individual bus transactions, and an emulation model as the instruction-set simulator (**ISS**) is running in the ARM7TDMI processors by inserting dummy computation workloads into the specification modules mapped to the processors. The **commPAM** and **commTLM** model are two different cases for the communication (*comm*) model, and the **ISS** is one of the implementation models.

	commTLM		commPAM		ISS	
	ref	distsim	ref	distsim	ref	distsim
sim time (sec)	4.27	2.81	44.98	25.28	174.04	87.43
timing	23757	23757	23749	23748	23749	23749
speedup	1	1.52	1	1.78	1	1.99
error	–	0	–	4.21e-5	–	0

Table 1: Simulation Results, for three TLMs of the JPEG Encoder examples

Table 1 shows the results simulated on two distributed PCs with Intel (R) Core(TM) 2 Duo CPU

E6550 at 2.33 GHz. Due to the tightly timed feature, the model cannot make good use of the multi-cores in one host. Nonetheless, an average speedup of **1.76** is gained by distributing the simulation onto 2 hosts. The heavier computation load the model has, the more speedup is gained by using the distributed simulator. Moreover, the timing error is very little for these three models with the relaxed timing synchronization.

5 Conclusion

In this paper, we have discussed the distributed DE simulation technique for the MPSoC models described in SLDLs. The inter-host communication channel is designed for the communication and timing synchronization between the distributed simulation hosts. The scheduling kernel is extended for time advancement handling and a lister thread is introduced for the relaxed timing synchronization and proper message handling. We have shown two sets of experiment: an un-timed basic pipeline example as well as a real-world application, JPEG encoder refined by SCE. Great speedup with scale to the number of simulation hosts is gained by using our distributed simulator. However, the imprecision of the timing due to the relaxed synchronization is not discussed yet. Our proposed distributed DE simulator suits better for simulating either the un-timed or approximate-timed transaction-level models.

Future work includes 1) automatic analysis algorithm and instrumentation for model partitioning. 2) timing error boundary analyzing and error correction during synchronizations.

Acknowledgment

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Lucai Cai, Junyu Peng, Chun Chang, Andreas Gerstlauer, Hongxing Li, Anand Selka, Chuck Siska, Lingling Sun, Shuqing Zhao, and Daniel D. Gajski. Design of a JPEG encoding system. Technical Report ICS-TR-99-54, Information and Computer Science, University of California, Irvine, November 1999.

[2] K.Mani Chandy and Jayadev Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452, Sept 1979.

[3] Weiwei Chen, Xu Han, and Rainer Dömer. ESL Design and Multi-Core Validation using the System-on-Chip Environment. In *HLDVT'10: Proceedings of the 15th IEEE International High Level Design Validation and Test Workshop*, 2010.

[4] Bastien Chopard, Philippe Combes, and Julien Zory. A Conservative Approach to SystemC Parallelization. In *International Conference on Computational Science (4)*, pages 653–660, 2006.

[5] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory. Relaxing Synchronization in a Parallel SystemC Kernel. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 180 –187, 10-12 2008.

[6] David Richard Cox. RITSim: Distributed SystemC Simulation, Sept 2005.

[7] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel Gajski. System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008(647953):13 pages, 2008.

[8] Forte Design Systems. Forte. http://www.forteds.com/.

[9] Richard Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct 1990.

[10] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer, 2000.

[11] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer, 2002.

[12] Kai Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably Distributed SystemC Simulation for Embedded Applications. In *International Symposium on Industrial Embedded Systems, 2008.*, pages 271–274, June 2008.

[13] International Telecommunication Union (ITU). *Digital Compression and Coding of Continous-Tone Still Images*, September 1992. ITU Recommendation T.81.

[14] David Nicol and Philip Heidelberger. Parallel Execution for Serial Simulators. *ACM Transactions on Modeling and Computer Simulation*, 6(3):210–242, July 1996.

[15] Ezudheen P, Priya Chandran, Joy Chandra, Biju Puthur Simon, and Deepak Ravi. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Mario Trams. Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead. *Digital Force White Paper*, Feb 2004.