Techniques to Reduce Synchronization in Distributed Parallel Logic Simulation

Kai-Hui Chang[1], Wei-Ting Tu[2], Han-Wei Wang[2], Yi-Jong Yeh[1], and Sy-Yen Kuo[2]
[1]Avery Design Systems, Inc.
Andover, MA 01810
USA
[2]Graduate Institute of Electronic Engineering, Dept. of Electrical Engineering, National Taiwan Univ.
No.1, Sec. 4, Roosevelt Road, Taipei
Taiwan
sykuo@cc.ee.ntu.edu.tw

ABSTRACT
As the complexity of chip designs increase, simulation time also increases. Unit and variable delay simulation takes the most simulation time in IC design process; however, parallel processing performs inefficiently due to large amount of synchronization. In this paper, techniques to reduce the number of synchronization points in synchronous designs are proposed, and a partitioner to partition designs along flip-flop boundaries is also proposed so that these techniques can be employed on real designs.

KEY WORDS
Distributed Parallel Simulation and Synchronization.

1. Introduction

Simulation has long been used to verify the correctness of chip designs. As the complexity of chips increase, longer and longer time is spent on simulation. It not only increases the cost of the chip but also delays its appearance in the market. Another problem that arises with larger chips is memory capacity. It is sometimes not possible to simulate large designs due to insufficient memory.

Therefore, parallel logic simulation has been proposed to solve this problem. Through parallel simulation, computation workload can be distributed to several computers and thus accelerate simulation. Since each computer simulates only part of the design, memory usage can also be reduced and memory capacity limitation can be removed.

There are two kinds of overheads in parallel processing: synchronization overhead and communication overhead. Partitioning plays an important role to reduce these overheads, and several algorithms have been proposed to solve this problem. For example, K-L [1] and F-M [2] have been successfully used to partition gate-level designs. However, they mainly focused on reducing communication overhead in gate-level designs and did not take synchronization overhead into consideration. Recent research like Manjikian and Loucks [3] also took other issues like network performance into consideration, but several issues are still not discussed, such as methods to reduce number of synchronization in practical designs under various kinds of clocks.

Synchronization problem is most serious in variable delay simulation, which usually occurs in post-synthesis simulation with Standard Delay Format (SDF) file annotated back. It takes the most time to simulate in the Integrated Circuit (IC) design flow. However, parallel simulation performs the worst in this level. It is because there are far more synchronization points than other levels of abstraction due to delays in gates and wires, and there is much less computation between synchronization points.

The most widely used and cheapest parallel processing environment is computers connected by Ethernet. Therefore, this paper focuses on characteristic of Ethernet, and several synchronization models and related issues that occur frequently in real designs are discussed. The way to reduce number of synchronization for real designs is proposed, and an algorithm which partitions the design along flip-flop boundaries is also proposed so that the technique can be used.

2. Background

2.1 Levels of Abstraction

Top-down design flow is the prevalent methodology used in current IC design process. The abstraction levels for logic design can be illustrated in Table 1 [4]. We apply our algorithm in gate level.

Abstraction	Primitives	Simulators
Functional level	Functional units (ALU, CPU)	Verilog, VHDL
Register transfer level (RTL)	Register, counter, MUX	Verilog, VHDL, HILO, THOR
Gate level	Gate, flip-flop, memory cell	Verilog, VHDL, HILO, THOR

Switch level	Ideal transistor (switch)	Verilog, VHDL, ESIM, COSMOS
	Transistor, resistor, capacitor	Verilog, VHDL, RSIM, RNL
Circuit level	Resistors, capacitor, current, voltage	SPICE, CAZM

Table 1. Abstraction levels for logic design.

From empirical studies on real designs, it is found that for each level of detail the total execution time grow by a factor of ten [5][6].

There are different timing models to model the temporal behavior of circuits:

Zero-delay assumes that every change of a signal's value will affect its output immediately.

Unit delay assumes that every change of a signal's value needs exactly one time unit to become available. This means a change at an input of an element is reflected in the next step on the output of that element.

Variable delay provides the most flexible way to simulate elements. It is used at lower levels and allows different switching times due to changing fanouts or capacitances that depend on the actual state of the simulated system.

2.2 Parallel Simulation Scheme

Methods of parallelizing logic simulation can be categorized into two major approaches: synchronous and asynchronous. In this paper, Avery Design System's SimCluster [7] is used. Depending on the synchronization method used, it can be either a synchronous or a conservative asynchronous parallel simulation environment.

2.3 Synchronization Models

There are at least three kinds of synchronization models:

Simulation backplane: It allows models to be accurate to basic time unit level. This provides the time-unit to time-unit accuracy but with very heavy overhead. This is normally used to catch those timing dependencies and gate-level ordering. It can also provide the hook to link different simulators, like analog and digital. This model uses delta-cycle synchronization and guarantees the most detailed timing accuracy.

Bus transaction model: It allows models to be accurate at each clock cycle level. This is consistent with the RTL description. What happens at time-unit level is not important at this level as long as there is no ordering dependency to affect results at cycle-level. This model uses cycle-based synchronization.

Data transaction model: At this level, it will synchronize at transaction level. For example, in a computer model, the CPU and DMA work in parallel. When a DMA is instructed to transfer a block of data to memory, there is no more interaction with the CPU. Each transaction may last from a few instructions to thousands of instructions. This model uses transaction-based synchronization.

3. Techniques to Reduce Simulation Overhead

The communication architecture used in Simcluster is TCP/IP and the time to send the packet is:

$$T_{packet} = T_{setup} + N \times T_{byte}$$

Where T_{packet} is the total time to send a packet, T_{setup} is the time to setup the packet until the first byte of the data can be sent. N is the number of bytes to be sent, and T_{byte} is the time to sent an extra byte in the packet. In Ethernet architecture, T_{setup} is much larger than T_{byte}, so the overhead to send an extra data in a packet is small compared with sending an extra packet[8].

In cycle-based synchronization, Cycle Per Second (CPS) of single process simulation, where the second is wall-clock second, is a good metric to determine if parallel simulation can be helpful. In Ethernet environment, assume there are N partitions, and two packets are sent between the process coordinator and each partition at a synchronization point, the time to finish one round of synchronization is $N \times T_{setupp} \times 2$ seconds. So the upper bound of the CPS that parallel simulation can be helpful is:

$$\frac{1}{N \times T_{setup} \times 2}$$

It is the maximum number of synchronization that can be done in one wall-clock second. If the CPS of single process simulation is smaller, there will be a better chance that parallel simulation can be helpful.

3.1 Principles to Reduce Overhead on Ethernet

In a distributed simulation environment on Ethernet, a one-byte socket write has much higher overhead than writing an extra byte in a packet. Therefore the following principles can be used as guidelines to design synchronization protocols and partition algorithms:

1. All port changes should be accumulated and be written in a socket write.
2. Number of synchronization should be reduced as many as possible, while number of ports between partitions is not so important. Therefore the partitioner has more freedom to duplicate or arrange instances in order to reduce synchronization and balance workload without concerning too much on connection between partitions. It is very different from traditional partitioning algorithms which minimize communication between partitions.
3. Synchronization using higher level of abstraction

should be used whenever applicable to reduce number of synchronization and increase concurrency.

3.2 Synchronize at Clock Edges

In order to guarantee correct simulation, the whole simulation must synchronize at the time that any partition has an event. However, the event may not cause port changes at the partition boundary and thus may not affect other partitions. In this case, the synchronization point is redundant. Determining the next synchronization point in advance is called "conservative lookahead." Its theory and impact on performance have already been studied by several researchers such as Nicol [9], Peterson and Chamberlain [10]. Therefore in this paper, we will focus on methods to determine future synchronization time in real designs instead.

Most of current designs are synchronous, and it provides good hints to determine the next synchronization point in advance. In synchronous designs, there are flip-flops between two blocks of circuits, and values propagate to another block only at clock edges. Therefore in synchronous designs, we can use cycle-based synchronization instead of delta-cycle synchronization to avoid redundant synchronization points.

To achieve this, we must determine the time of the next clock edge in advance. There are two approaches to predict the next clock. One is presimulation, and the other is dynamic phase locker.

3.3.1 Presimulation. Presimulation requires simulating the design in single process or multiple-processes with delta-cycle accuracy first and write clock changes to a Value Change Dump (VCD) file. Then this information can be used to determine the time of the next clock edge for cycle-based simulation.

This approach may seem to be naïve at the first glance; however, it is practical in real designs due to the following reasons:

1. Usually, only the clock generator needs to be presimulated and its simulation time is short.

2. For clocks that are not periodic, like spread-spectrum clocks, this approach is the only way to know the time of the next clock edge in advance.

3.3.2 Dynamic Phase Locker. Dynamic phase locker tries to lock periodic clocks during simulation. Once a clock pattern repeats more than three times, the clock is locked and the same pattern is used to determine the next clock time. And then cycle-based synchronization will be started. If the period of a clock changes, it will be unlocked, and the simulation will return to delta-cycle synchronization immediately. This approach is more flexible and is adaptive to clock changes. However, it only works if the clock is periodic.

There are two modes: global locked and global unlocked. When all clocks are locked, the system will enter global locked mode. In this mode, next synchronization time is determined from clock patterns saved. Otherwise the system is in unlocked mode, and delta-cycle synchronization is used.

4. Clock Partitioner

Gate level simulation with variable delay is the most time-consuming one in the IC design process. The netlist of gates is usually produced by a synthesis tool, and the temporal behavior is calculated by timing tools. The design is often flattened at this abstraction level, and no hierarchical information is preserved. However, the performance of distributed simulation is usually poor in these designs because complicated timing will result in lots of synchronization points, and the workload between synchronization points will be low. In this situation, the synchronization overhead is too much and will slow simulation down. But if the design is partitioned at flip-flop boundaries, only synchronization points at clock edges will be necessary. The number of synchronization points will then be reduced significantly, and timing accuracy is preserved because it is guaranteed that no events occurred between clock edges will cause port changes on partition boundaries.

In order to have correct simulation results, the simulation should synchronize at the following time:

1. When clock changes, in order to propagate clock signal.

2. The time that the output of the flip-flop changes. It is the time that the clock changes plus the propagation delay from clock to the output of the flip-flop.

If more than one kind of flip-flops is used at flip-flop boundary, there may be more than one propagation delay and the ports may change at different times. In this case, synchronization points should be added whenever the port values change. However, using the largest delay is enough for most designs.

Mueller-Thuns et al [11] proposed an approach specifically for sequential logic circuits which produces partitions such that only latch outputs cross partition boundaries. However, their approach may need to duplicate gates for different partitions and is less efficient when no such operation is allowed or required. Therefore a new algorithm to find flip-flop boundaries is proposed in this paper, called clock partitioner. It is very efficient since most gates will be processed only once, except that flip-flops may be processed twice.

4.1 Algorithm

i, j are instances in the design. Three functions are used to return the ports of an instance: input(i) returns its inputs, output(i) returns its outputs, and ports(i) returns its ports.

In the algorithm, ports for clocks are not counted as input ports.

U is the set of instances not partitioned. Initially, it contains all the instances in the design.

P contains instances added to the partition and is initially empty.

S is the set of storage devices in the design.

PORT is the set of instances that will become ports at partition boundary. It is initially empty.

Q is a queue used during processing. Initially, it contains an instance selected randomly from *U*.

"Connect" is a function which returns true and marks flags on the arguments if the two instances in its arguments are connected.

"marked" returns true if flag is marked on the port.

"Dequeue" returns the first instance in the queue and removes it from the queue.

The algorithm is given in Figure 1.

```
1.    while Q ≠ null do
2.      i ← dequeue(Q)
3.      if i ∈ S then
4.        if i ∉ PORT then
5.          PORT ← PORT∪ {i}
6.          if !marked(output(i )) then
7.            Q ←Q ∪ {j ; connect(input(i ), j ) &&
                        j ∈ U && j ∉ Q}
8.          else if i ∈ PORT && marked(port(i )) then
9.            PORT ←PORT–{i}
10.           P ←P ∪ {i}
11.           U ←U - {i}
12.           Q ←Q ∪ {j ; connect(port(i ), j ) &&
                        j ∈ U&& j ∉ Q}
13.     else
14.       P ←P ∪ {i}
15.       Q ←Q ∪ {j; connect(port(i), j)&& j ∈ U &&
                                        j ∉ Q}
```

Figure 1. Clock Partitioner Algorithm

In line 2, an instance *i* is removed from the queue, *Q*. In line 5, if *i* is a storage device, and it is not in *PORT*, it is added to *PORT*. In line 6-7, if *i*'s output ports are not marked, then all instances connected to *i*'s inputs not in the queue and not partitioned are added to the queue. In line 8-12, if *i* is a storage device, is already in the *PORT*, and has its input and output ports marked, then it is a flip-flop inside a partition and is removed from *PORT* and added to the partition *P*. All unpartitioned instances connected to its ports are also added to the queue. In line 13-15, the instance *i* is not a storage device. So it is added to *P* directly. All unpartitioned instances connected to it and are not in the queue are also added to the queue.

A partition bounded by flip-flops will be produced by this algorithm and is stored in *P*. To partition the whole design, this algorithm can be repeated until *U* is empty. Flip-flops on the partition boundary will be added to the partition that connects to their inputs.

This algorithm may also find a partition which contains only primary input and output ports and does not connect to other partitions. In this case, that partition can be simulated independently; otherwise delta-cycle synchronization must be used.

In this algorithm, each instance is processed once or twice (twice when it is a storage device inside a partition), so its time complexity is O(n).

4.2 Clock and Flip-flop Recognizer

To use clock partitioner, flip-flops and clocks must be known in advance. The user can either specify them explicitly or use the clock and flip-flop recognizer provided with the partitioner.

The recognizer tries to recognize flip-flops in the cell library. In the recognizer, it is assumed that the clock is always in the port list. If the cells are modeled in RTL level, the port in "always @(port)" construct is recognized as a clock port. If the cells are modeled in gate level and specify blocks exist, the "posedge/negedge port" used in $setup construct is recognized as clock port. If a clocked port is found in a cell, the cell is recognized as a flip-flop.

5. Experimental Result

In Simcluster's distributed simulation model, there is a process coordinator, called "master," which handles signal transfer and synchronization among partitions. All other partitions are called "children" and are coordinated by the master. The experimental design is a 128 * 128 flattened gate-level multiplier and its gate count is 8832. It is composed of four Wallace tree multipliers and three ripple-carry adders. The design is three-stage pipelined as Figure 2 shows and is partitioned by clock partitioner. After partitioning, the multipliers are simulated by children, and the testbench and adders are simulated by the master.

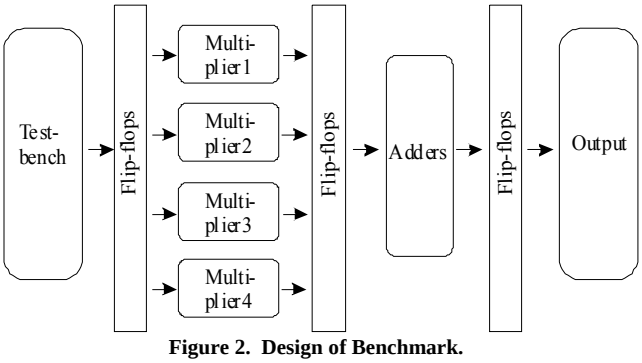

Figure 2. Design of Benchmark.

Simulation environment:
Simulator: VCK
Platform: Redhat Linux 8.0
Partitioner run time: 2.9 seconds

Simulation is distributed to five AMD XP 1.8GHz computers with 512M RAM connected by local area network (100Mbps Ethernet) with five partitions.

Result:

Delta cycle synchronization result is given in Figure 3, and synchronization at clock edge result is given in Figure 4.

In the figures, Var-delay means variable delay.

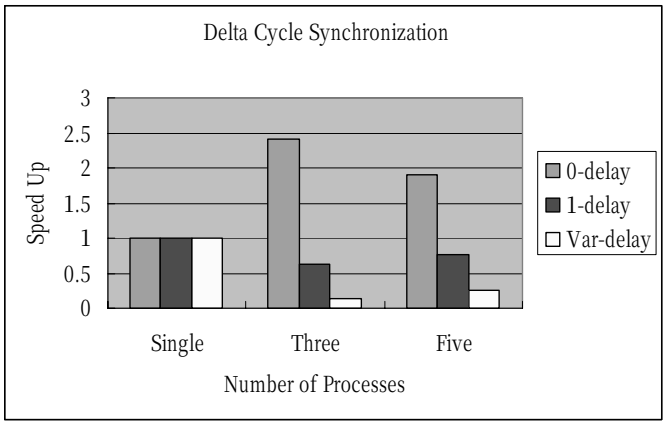

Figure 3. Delta Cycle Synchronization Results.

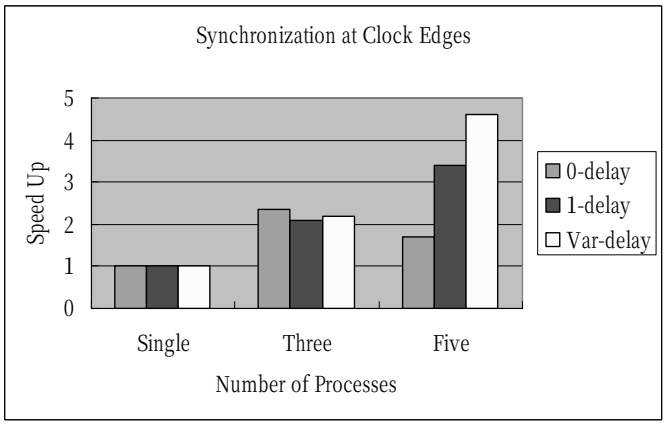

Figure 4. Cycle Based Synchronization Results.

From Figure 3, we can find that the speed up of 3 or 5 partitions is better than that of a single process. That is, the simulation is benefited from parallel processing. However, the speed up of five partitions is less than that of three partitions because the communication overhead acts against the parallel processing speed up. Next, both unit-delay and variable-delay simulations with more than one partitions do not have any speed up due to large amount of communication overhead. Third, variable delay simulation is slower than unit delay simulation due to extra synchronization produced by the complex timing in variable delay simulation.

From Figure 4, we can find that both unit-delay and variable delay simulations are benefited from parallel processing and the speed up is almost proportional to the number of processes.

From the experimental results, we can see that communication overhead plays an important role in the performance of parallel processing. Complex timing models introduce too much synchronization overhead and make delta-cycle synchronization impractical to simulate designs with unit or variable-delay. Furthermore, synchronization at clock edges reduces communication overhead significantly and the speed up is almost linear to the number of processors used, which makes parallel processing applicable to real designs.

6. Conclusion

In this paper, techniques to determine lookahead synchronization time by exploiting clocks in synchronous designs are described. A partition algorithm with time complexity O(n) is also proposed to partition the design at flip-flop boundaries so that these techniques can be used. From the experimental results, it can be concluded that for unit and variable delay models, parallel distributed simulation synchronized at delta-cycle accuracy is impractical because of large amount of communication overhead. Besides, our techniques can indeed improve performance and make speed up almost linear to the number of processes. Techniques proposed in this paper can reduce simulation time significantly, especially in post-synthesis designs with variable delay. They will greatly save the simulation time.

References:

[1] W. Kernighan and S. Lin., An Efficient Heuristic Procedure for Partitioning Graphs, Bell System Technical Journal, 1970

[2] C. M. Fiduccia and R. M. Mattheyses, A Linear-Time Heuristics for Improving Network Partitions, *Proceedings of the 19th Design Automation Conference*, 1982

[3] Naraig Manjikian and Wayne M. Loucks, High Performance Parallel Logic Simulation on a Network of Workstations, *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, 1993

[4] Gerd Meister, A Survey on Parallel Logic Simulation, Dept. of Computer Science, University of Saarland, Germany, Sep. 1993

[5] L. Soulé, and T. Blank, Statistics for Parallelism and Abstraction in Digital Simulation, *Proc. 24th Design Automation Conference*, 1987

[6] K. Wong, M. Franklin, R. Chamberlain, and B. Shing, Statistics on Logic Simulation, *Proc. 23th Design Automation Conference*, 1986

[7] Avery Design Systems, *VCK/Simlib User's Guide, Rev. 1.1.0*, May 2002

[8] M. Hassan and R. Jain, *High performance TCP/IP NETworking Concepts, Issues and Solutions*, Pearson Prentice Hall, 2004

[9] D. M. Nicol, High Performance Parallelized Discrete-Event Simulation of Stochastic Queuing Networks, *Proceedings of Winter Simulation Conference*, 1988

[10] George D. Peterson and Roger D. Chamberlain, Exploiting Lookahead in Synchronous Parallel Simulation, *Proceedings of Winter Simulation Conference*, 1993

[11] Mueller-Thuns, R. B. Saab, D. G., Damiano, R. F., and Abraham, J. A., Portable Parallel Logic and Fault Simulation, *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1989