# AUTOMATIC PARTITIONER FOR DISTRIBUTED PARALLEL LOGIC SIMULATION

Kai-Hui Chang[1], Han-Wei Wang[2], Yi-Jong Yeh[1], and Sy-Yen Kuo[2]

[1]Avery Design Systems, Inc.
Andover, MA 01810, USA

[2]Dept. of Electrical Engineering, National Taiwan Univ.
Taipei, Taiwan

changkh@avery-design.com.tw, andy@lion.ee.ntu.edu.tw, yeh@avery-design.com.tw, sykuo@cc.ee.ntu.edu.tw

**ABSTRACT**

As the complexity of circuit design increases, verification through simulation has become the bottleneck of the IC design process. Distributed parallel simulation is one way to solving the problem. In order to distribute the simulation to multiple processors, the design must be partitioned first. This paper describes the goal and criteria for a distributed simulation partitioner and proposed an algorithm to achieve it. While most existing partitioners focus on gate-level, this partitioner is designed for all levels of abstraction, from behavior level to gate level.

**KEYWORDS**

Partitioning, distributed, parallel, and logic simulation

## 1. Introduction

As the complexity of Very Large Scale Integrated Circuit (VLSI) design increases, verification through simulation has become the bottleneck of chip design process. Large designs, such as System on Chip (SOP), may take several days to simulate. Sitting for simulation to finish is a waste of time and increases the chip's time to market. Therefore Field Programmable Gate Array (FPGA) prototyping, such as emulator, is proposed to solve this problem. However, emulators are expensive and hard to use.

Memory usage is another problem that arises when the design gets large. The physical memory of a computer is limited, so it may not be possible to simulate large designs that exceed the computer's capacity of memory.

Therefore some Electronic Design Automation (EDA) tool vendors provide distributed simulation solutions to solve the problem, such as Avery Design Systems' Simcluster [1]. Through distributed simulation, the same workload can be distributed to several processors and improve the turnaround time of simulation. By partitioning a design to several smaller pieces, each piece takes less memory and can easily fit into any current computer. However, partitioning is still a problem for these tools.

The goal of the partitioner is to make the workloads among processors as balanced as possible and make the communication overhead as small as possible. Several algorithms have been proposed and used extensively. For example, K-L [2] and F-M [3] have been successfully used to partition gate-level designs. Recent researches by Prathima [4], Patil *et al* [5], Smith [6], and Subramanian [7] took other issues into consideration like delay, communication and concurrency. These partitioners further improved the quality of partitioning.

However, most of them focus on gate-level partitioning and no single algorithm works well for all abstraction levels of a design. Therefore in this paper, a partitioner that handles all abstraction levels is proposed. It can partition a design from behavior level to gate level. Other related issues are also discussed.

There are different approaches for parallelizing logic simulation, such as synchronous and asynchronous [8]. Avery Design Systems' Simcluster is a synchronous parallel simulation environment. Simcluster supports Verilog and Very High Speed IC Hardware Description Language (VHDL) designs. In this paper, the partitioner is focused on designs written in Verilog.

In Simcluster's distributed simulation model, the original design is partitioned into several pieces called partitions. There is a process coordinator, called "master," which handles signal transfers and synchronization among partitions. All other partitions are called "children" and are coordinated by the master. "Master" is also a Verilog simulator and can simulate part of the design.

The algorithm for distributed simulation is given below.

1) Each child updates its input/inout ports from master, simulates its circuit, and then sends port changes to master.
2) Master collects port changes. If there are port changes for a child, send those changes to that child and ask that child to do step 1 again. If no port change has been collected, go to step 3.
3) Master asks all children to go to read-only synchronization phase, and then each child sends its next event time.
4) Master collects all the next event time from its children and calculates the minimum one; it is the next event time for all children. Master then sends the time to all children, and all children will advance their simulation time to the agreed next event time and then go to step 1.

Parallel processing is achieved in step 1 and 2, where all children simulate their own circuits concurrently. There is overhead from step 3 and 4, where synchronize signals are

sent, and next event time is collected and sent back to each child. Another overhead comes from propagating port changes from one child to another child. It is because in distributed simulation, socket reads/writes are used, and they take a much longer time than the time used to propagate port changes in single process. For better performance in distributed simulation, workload between two synchronization points must be high enough to compensate these overheads.

## 2. Partitioner Modes

In this paper, the partitioning problem for distributed simulation is mapped to a graph problem. The design is converted to a tree, and the root is the top module. The nodes of the tree are instances of the design. The children of an instance are connected to their parent by branches. Each node has two weights, self-weight and all-weight. Self-weight is the workload of the instance, and all-weight is the workload of the subtree rooted at that instance. The goal is to select a certain number of nodes from the tree and make all-weight of the nodes as close to balanced workload as possible.

### 2.1 Normal Mode

In normal mode, the design hierarchy is preserved. Only the nodes in the tree can be selected. Consider the tree in Figure 1: (The number in the parentheses is the workload)

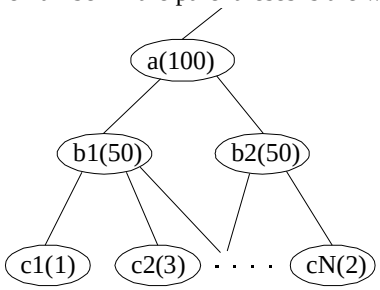


**Figure 1. Example normal mode design hierarchy.**

If b1 and b2 are chosen to be partitions, the workload will be balanced.

### 2.2 Regroup Mode

In some designs, especially those with wide and shallow hierarchies, good partition results cannot be produced in normal mode. For example, in Figure 2, neither the leaf nodes "cN" nor the middle node "a" can be selected. Because the weight of "cN" is too small, and the weight of "a" is too large. However, such a case is common in real designs. After a design is synthesized, it may become wide and shallow.

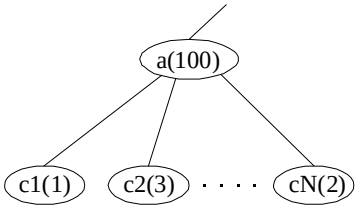


**Figure 2. Example regroup mode design hierarchy.**

Therefore a new partitioner mode, regroup mode, is proposed to solve this problem. In regroup mode, instances like "cN" will be regrouped and another level of hierarchy will be added between "a" and "cN" for the new groups, as Figure 3 shows. In the Figure, "b1" and "b2" are the new instances added to regroup "cN". Good partition results can thus be obtained.

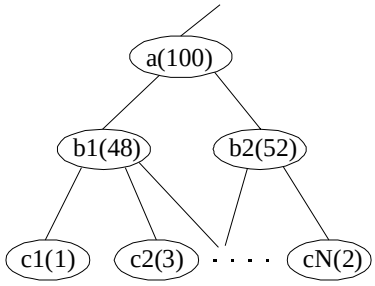


**Figure 3. Example design hierarchy after regroup.**

### 2.3 Design Hierarchy Flatten

Workload among partitions can be further balanced if the instances to be partitioned are more fine grained. In Figure 4, suppose that all instances, cN, in level 3 have workload 2. Now we want to partition the instances at the second level into two partitions, then we can only choose b1 and b2 as the result partitions, and the workload will be unbalanced. But if we flatten the second level and partition the third level directly, we can partition c1 to c15 to the first partition, and partition c16 to c30 to the second partition. In this way, the workload will be balanced.

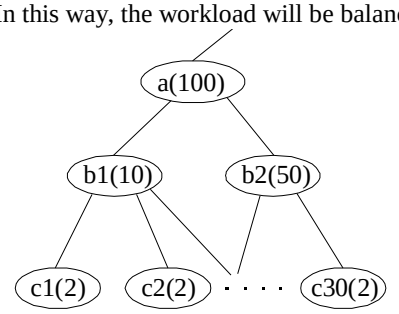


**Figure 4. Example design hierarchy for flattening.**

## 3. Problem Formulation

### 3.1 Preliminaries

*1) Partition pattern:* A set of legal distributed simulation partitions is called a partition pattern.

*2) Variable and net:* The difference between a variable and a net is that a variable can store a value, while the value of a net is determined by its driver.

### 3.2 Goal

The purpose of the distributed simulation partitioner is to make workloads among different processors as balanced as possible and to make communication overhead as small as possible. The partitioner should read the Verilog design and output the top-modules of partitions as its result.

### 3.3 Cost Function

Since the goal of the partitioner is to make workload in each partition as balanced as possible, the best partition result is to make each partition have the following

workload, which is defined as "goal workload."

$$\text{Goal workload} = \frac{\text{total workload}}{\text{number of partitions}}$$

In order to have a better partition result, the deviation of each partition's workload to the goal workload should be as small as possible. Therefore the cost of a node is defined as the following in normal mode.

$$\text{cost} = \left|\text{all weight - goal workload}\right| + \text{additional cost}$$

Additional cost takes communication overhead due to remote access into consideration. A remote access is a hierarchical reference of variable that crosses partitions. More remote accesses usually mean more communication overhead. A parameter specified by the user, "remote_access_factor", is used to add cost to each remote access. The larger the number is, the more cost it will add. It should range between 0.0 and 1.0, and its default is 0.01. The additional cost due to this parameter is calculated by the following formula.

Additional cost=goal workload * remote_access_factor

See Figure 5 for the graph representation of the cost function. Additional cost is not shown in the Figure.

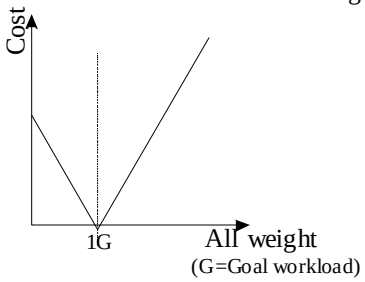


**Figure 5. Normal mode cost function.**

If the design hierarchy does not need to be preserved, the user can use regroup mode for better partition results. In regroup mode, the cost function is changed to:

$$\text{cost} = \tfrac{1}{2}\times G - \left|W \,\%\, G - \tfrac{1}{2}G\right| + 0.1\times\left\lfloor \tfrac{2}{3}\times\tfrac{W}{G}\right\rfloor\times G + A$$

Where G = goal workload, W = all weight, A = additional cost. The graph representation of the cost function is shown in Figure 6.

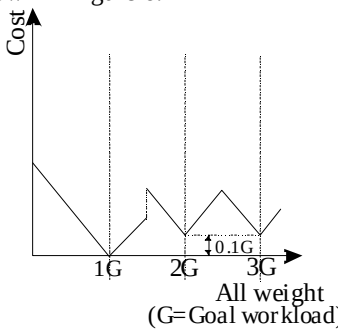


**Figure 6. Regroup mode cost function.**

The reason to add additional 0.1G when W >= 1.5G is to make the partitioner select nodes with W close to 1G first and avoid the use of group partitioner unless necessary.

If a node with "W >= 1.5G" is selected, regroup partitioner engine will be used to regroup the nodes under it to "W / G" partitions.

### 3.4 Workload Estimation

Workload in a simulation consists of processing signal changes, scheduling the affected behaviors, and executing the affected behaviors. There are two ways to estimate workload. One is static, and the other is dynamic. Static workload can be obtained by parsing the original design and does not capture any dynamic behavior. There are several ways to estimate this number such as gate count in a partition, number of VHDL processes (in Verilog, number of always machines) [9], and some of them take activity rate into consideration and includes sensitivity of each component [10]. In this paper, number of variables and nets in a partition is used as static workload. In Register Transfer Level (RTL) or behavior level, the idea of using the number of variables and nets is similar to using the gate count in gate level.

Dynamic workload can only be obtained if simulation has been run. It is "number of value changes of variables and nets in a partition." If there are more variables and nets in a partition, and if their values change more often, the workload of that partition will be higher. In RTL and behavior level, this definition gives a good estimation on the dynamic workload.

Dynamic workload is obtained by simulating the design with a tool provided with the partitioner, called Evaltool, for a certain period of time. The tool will sample the value changes of variables and nets in instances and calculate dynamic workload during simulation.

The user should run the tool during "typical" simulation period. Here "typical" means that it is representative for the workload distribution most of the time during simulation. For example, the circuits activated during initialization are usually different from those after initialization. However, initialization usually takes only a short period of time to run during simulation. Running Evaltool during initialization will then obtain an atypical workload distribution and may bias the partitioner.

After simulation finishes, the workload of every instance is written to a file and is used by the partitioner. Dynamic workload captures dynamic behavior of the circuit and is much more accurate than static workload.

## 4. Partitioning

### 4.1 Input and Output

The partitioner will parse Verilog source files and build a design hierarchy tree. The following parameters are used to control the partitioner.

*1) Min_partition:* The minimum number of partitions.

*2) Max_partition:* The maximum number of partitions.

*3) Max_deviation:* Here, deviation means the difference in workload between a partition and the goal workload. If the number is larger, larger deviations in workload is allowed in the final partition pattern. It should range

between 0.0 and 1.0.

*4) Remote_access_factor:* Remote access cost factor.

*5) Partition_number_factor:* More partitions usually bring more communication overhead and additional cost will be added by this factor.

*6) Flatten_instances:* Instances to be flattened for better partition results.

All these parameters are specified by the user.

The partitioner will select a certain number of nodes from the design hierarchy tree, and each node represents a partition. A code generator will generate auxiliary code for Simcluster integration and is not discussed in this paper.

### 4.2 Partition Algorithm

In the algorithm, a minimum heap is used to sort the nodes according to their costs. It is used to find the node with the smallest cost quickly.

1) Traverse the whole design hierarchy tree. Calculate the cost of each node and add the nodes to the heap.
2) Get the node with the smallest cost from heap.
3) Recalculate the costs of the nodes affected by this node and update the heap by the following rule: Subtract the all_weight of this node from the all_weight of all its ancestor nodes.
4) If N partitions are needed, repeat 2 and 3 N times.
5) Check the nodes selected. If max_deviation is violated, the partition pattern is abandoned.

The partition algorithm will be run from partition_number = min_partition to max_partition. The best partition result is the partition pattern with the smallest accumulated cost. Partition_number_factor is used here to add additional cost to each partition pattern. So the final cost is:

Cost× (1+ number of partitions × partition_number_factor)

### 4.3 Regroup Algorithm

This problem is similar to traditional partition problems: There are lots of gates, and the partitioner will produce partitions of gates that have similar workload and as small communication overhead as possible between them. A greedy algorithm is used here:

1) Create a partition.
2) Choose an instance as seed and add it to the partition.
3) Add all instances connected to the instance to the partition. If no instance connects to the instance, go to 2.
4) Repeat until the goal workload of the partition is exceeded.

These steps are repeated until all instances have been partitioned.

### 4.4 Flatten Algorithm

Hierarchy flatten is a bottom-up procedure. Instances on the lowest level are expanded up level by level until all hierarchies in between are flattened. Assume the hierarchy is A-B-C, and instance B will be flattened:

1) Based on port connections, replace the variables used in the ports of B by variables defined in A.
2) If there is behavior code like initial and always, copy it up to level A. If there are variables defined in B, they will not be replaced in step1. Copy them to A, too.
3) Repeat 1 and 2 until all levels in between are flattened.

### 4.5 Complexity Analysis

#### 4.5.1 Partition Algorithm

Space complexity:

A node is created for each instance, and a pointer is needed for each node in the heap. So if there are n instances in the design, the space complexity is O(n).

Time complexity:

It takes O(n) time to build the design hierarchy tree.

The time complexity to add, rearrange and remove a node in the heap is O(logn).

It takes O(nlogn) to build the heap, and it takes O(nlogn) to recalculate the costs and update the heap after a node is selected.

Since partition_number is a small constant number, the total time complexity of the algorithm is O(nlogn).

#### 4.5.2 Regroup Algorithm

Time complexity:

There are N instances to be partitioned. When an instance is selected, it takes up to N searches to find instances connected to it. Therefore the time complexity is $O(N^2)$.

## 5. Experimental Results

### 5.1 Benchmark 1

The design is a microprocessor from Aquarius project developed by Opencores [11]. The block diagram of the CPU is given in Figure 7.

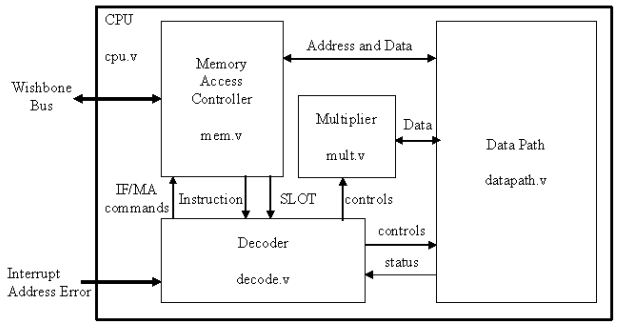


**Fig. 7. Block Diagram of Aquarius CPU.**

**Simulation environment:**
Platform: Redhat Linux 8.0
Simulator: VCK
Partitions: cpu.datapath, memory, cpu.decode
Partitioner run time: 0.3 seconds

**Results:**

| |
|---|
| Platform: Four Intel Xeon 2.2G |
| Single process simulation time: 512.18 seconds |
| Distributed simulation time: 216.31 seconds |
| Speed up: 2.37X |

### 5.2 Benchmark 2

The design is a 128 * 128 gate-level multiplier. It is

composed of four Wallace tree multipliers and three ripple-carry adders. The design is three stages pipelined as Figure 8 shows. After partitioning, the multipliers are simulated by children, and the testbench and adders are simulated by the master.

In this benchmark, the hierarchy has been flattened and regroup mode has been used.

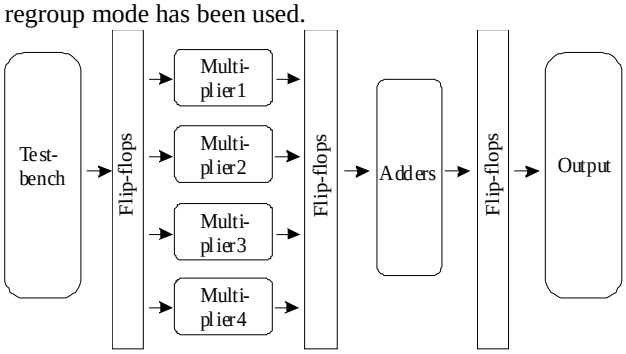


**Figure 8. Design of Benchmark 3**

**Simulation environment:**

Master: VCK

Single and child processes: Cadence Verilog XL 3.40

Platform: Redhat Linux 8.0, on five P-III 600 computers connected by 100M Ethernet

Partitioner run time: 45 seconds.

**Results:**

| |
|---|
| Single process: 261 seconds |
| Distributed: 82 seconds |
| Speed up: 3.18X |

## 6. Conclusion

In this paper, the goal and criteria of partitioning for distributed simulation is described, and a partitioner is designed to solve the problem. It supports two partitioner modes – normal and group mode, and utilizes techniques to flatten the design for finer grained partitioning. Unlike other partitioners, which usually focus on gate-level circuits only, this partitioner can partition any design in any level of abstraction, as long as it is written in Verilog. From the results of the benchmarks, it can be concluded that the partitioner purposed in this paper can indeed find good partition pattern that accelerates circuit simulation by distributed simulation. With this tool, distributed simulation tools will become easier to use and will greatly save circuit designers' and verifiers' time.

**REFERENCES**


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

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

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

[4] P. Agrawal. Concurrency and Communication in Hardware Simulators, *IEEE Transactions on Computer-Aided Design*, Oct. 1986

[5] S. Patil, P. Banerjee, and C. D. Polychronopoulos, Efficient Circuit Partitioning Algorithms for Parallel Logic Simulation. In *Proceedings, Supercomputing, '89*, Nov. 1989, 361-370

[6] S. P. Smith, B. Underwood, and M. R. Mercer, An analysis of several approaches to circuit partitioning for parallel logic simulation, *Proceedings of the 1987 International Conference on Computer Design*, IEEE, New York, 1987, 664-667

[7] Swaminathan Subramanian, Dhananjai M. Rao, and Philip A. Wilsey, Study of a Multilevel Approach to Partitioning for Parallel Logic Simulation, *Parallel and Distributed Processing Symposium*, 2000

[8] Gerd Meister, A Survey on Parallel Logic Simulation, University of Saarland, Germany, Sep 1993

[9] Kevin L. Kapp, Thomas C. Hartrum, and Tom S. Wailes, An Improved Cost Function for Static Partitioning for Parallel Circuit Simulations Using a Conserative Synchronization Protocol. *Parallel and Distributed Workshop*, 1995

[10] Pavlos Konas and Pen-chung Yew, Partitioning for Synchronous Parallel Simulation, *Parallel and Distributed Simulation Workshop*, 1995

[11] http://www.opencores.org/projects/aquarius/