

A design-driven partitioning algorithm for distributed Verilog simulation

Lijun Li and Carl Tropper
School of Computer Science
McGill University
Montreal, Canada
lli22, carl@cs.mcgill.ca

Abstract

Many partitioning algorithms have been proposed for distributed VLSI simulation. Typically, they make use of a gate level netlist, and attempt to achieve a minimal cut size subject to a load balance constraint. The algorithm executes on a hypergraph which represents the netlist.

In this paper we propose a design-driven iterative partitioning algorithm for Verilog based on module instances instead of gates. We do this in order to take advantage of the design hierarchy information contained in the modules and their instances. A Verilog instance represents one vertex in the circuit hypergraph. The vertex can be flattened into multiple vertices in the event that a load balance is not achieved by instance based partitioning. In this case the algorithm flattens the largest instance and moves gates between the partitions in order to improve the load balance.

Our experiments show that this partitioning algorithm produces a smaller cutsize than is produced by hmetis on a gate-level netlist. It produces better speedup for the simulation because it takes advantage of the design hierarchy.

1 Introduction

Modern VLSI systems are becoming increasingly complex, posing a never-ending challenge to sequential simulation. In order to accommodate the growing need for increased memory as well as the need for decreased simulation time, it is becoming increasingly necessary to make use of distributed simulation[24].

Time Warp[12] is an appealing technique for the distributed logic simulation of VLSI circuitry because it can potentially uncover a high degree of parallelism in the VLSI system being simulated.

However, getting satisfactory simulation performance in a distributed environment is challenging since we need to overcome the huge cost of inter-processor communication which is exacerbated in a distributed environment by

netlists comprised of millions of gates. It is widely known that partitioning is an NP-complete problem, the result of which is that partitioning algorithms provide heuristic solutions and can be trapped in local minima.

Most of the partitioning algorithms [21, 9, 1, 18, 22, 15, 20, 11, 14] for distributed/parallel VLSI simulation partition gate level netlists. These algorithms are typically used for floorplanning and placement, not for simulation. They can produce a big cutsize which is intolerable in a distributed VLSI simulation environment because of the communication costs which are a consequence of a large cutsize. Moreover, few partitioning algorithms take load balancing into account.

The ASIC design community has a well-established hierarchical design methodology. Every design is partitioned into blocks by functionality. The design hierarchy is reflected in modules and their instances in Verilog[23]. In this paper we take advantage of the design hierarchy information present in Verilog and combine it with a move-based partitioning algorithm. In our algorithm, the module/instance is the basic partitioning element instead of the gate.

The rest of this paper is organized as follows. Section 2 is devoted to related research. In section 3, we introduce hierarchy in Verilog. Our distributed simulation environment DVS[16] is briefly described in section 4. In section 5, we present the details of our design-driven partitioning algorithm. A comparison of the cutsize and of the execution time of our design-driven partitioning algorithm and hmetis partitioning based on netlists is presented in section 6. The last section contains our conclusions and thoughts about future work.

2 Related work

[5] proposes an architecture driven partitioning algorithm for netlists with multiterminal nets. The target architecture was a multifield-programmable gate array (FPGA).

The goals of the algorithm are to minimize the number of FPGA chips used and to maximize routability.

[17] uses a module-based simulation mapping method. Although the details of the algorithm are not described in the paper, the author states that it reduces the communication cost and achieves a better load balancing.

[4] uses the module tree as the data structure instead of the circuit hypergraph. Modules are not moved by the algorithm. Nor does it use an iterative improvement technique. The author does not mention the cutsizes achieved by the algorithm and concludes that the algorithm achieves better performance than a sequential simulation and is efficient.

[6] proposes a module migration based partitioning algorithm which tends to keep the cluster intact in order to reduce the net cut size. The algorithm implicitly promotes the move of clusters of modules during the module migration process by paying more attention to the neighbours of moved modules, relaxing the size constraints temporarily during the migration process, and controlling the module migration direction. Load balancing was not considered in this algorithm.

Iterative algorithms start from an initial partitioning and try to improve it. The well-known iterative algorithms for circuit partitioning are CLIP/CDIP[20], Metis/hMetis[13] and F-M[9]. It is worthwhile noting that the CLIP[20] algorithm tries to detect and restore the cluster destroyed by the iterative partitioning algorithm based on the flattened netlist.

[13] introduces a coarsening phase in a multilevel hypergraph partitioning algorithm. During the coarsening phase, a sequence of successively smaller hypergraphs is constructed. The purpose of coarsening is to create a smaller hypergraph while preserving the partitioning quality obtained from the original hypergraph. The authors claim that hmetis produces partitions that are consistently better than other widely used algorithms and is one to two orders of magnitude faster than other algorithms.

[20] and [13] try to reduce the size of the hypergraph from the bottom up, i.e. they extract clusters from the flattened netlist without worsening the quality of the partitioning of the original netlist. Our algorithm works from top-to-bottom-it flattens the design hierarchy step by step and compromises between the load balancing constraint and the minimum cutsizes.

Iterative algorithms generally work on any hypergraph while our algorithm specifically targets distributed Verilog simulation. The main purpose of our algorithm is to try to keep the Verilog instance (actually the design hierarchy) intact from the beginning. It is much easier than restoring it from the debris produced by first flattening the netlist. Moreover, the quality of the resulting partition should be better than the cluster restoration and hypergraph coarsening.

3 Hierarchy in Verilog

The module is the basic unit of code in the Verilog language. Both behavioral and structural code can be contained within a module. The encapsulation property of the module gives designers the ability to reuse the module in a VLSI design. Moreover, the module provides an interface to the program while hiding the complexity inside of it. Therefore the module and its instance are natural candidates for partitioning. We introduce the concept of a super-gate in this paper in order to describe the module instance in a circuit hypergraph.

Modules can reference lower level modules and describe the interconnections between them as part of the hierarchy. Each module instance is an independent, concurrently active copy of a module. It contains the name of the original module, an instance name that is unique to that instance (within the current module) and a port connection list.

Usually Verilog module instances communicate with other instances through ports. The encapsulation property of Verilog modules helps to achieve a smaller cutsizes when we partition the circuit. Although Verilog supports cross module reference, standard design practice discourages such usage.

Figure 1 shows a design hierarchy described by Verilog. The left side of the figure is the Verilog source code while the right side displays the design hierarchy and its interconnection. Coupling is usually loose between Verilog instances and is tight inside a Verilog instance (at least for a good VLSI design). Therefore, if the circuit is cut at instance boundaries, the cutsizes will be smaller and inter-processor communication will be reduced.

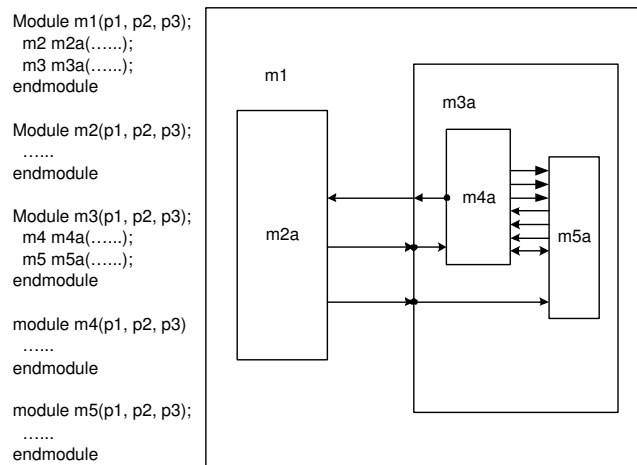


Figure 1. Verilog module/instances and inter-connection

We should note that not only does RTL(Register Transfer

Level) Verilog source code contain design hierarchy information, but the synthesized gate level design also contains exactly the same design information. The design information is lost after elaboration, a process to flatten the design hierarchy. However, if partitioning is done before elaboration we are able to take advantage of the design information.

4 DVS[16]: A framework for distributed Verilog[23] simulation

Before we present the implementation of our algorithm we present a brief description of DVS, a framework for distributed Verilog simulation. Several kinds of partitioning algorithms are implemented in DVS.

Figure 2 portrays the architecture of DVS. The 3 layers of DVS are shown on the right side of figure 2. The bottom layer is the communication layer, which provides a common message passing interface to the upper layer. Inside this layer, the software communication platform can be PVM or MPI. Users can chose one of them without affecting the code of upper layers.

The middle layer is a parallel discrete event simulation kernel, OOCTW, which is an object-oriented version of Clustered Time Warp (CTW)[2]. It provides services such as rollback, state saving and restoration, GVT computation and fossil collection to the top layer.

The top layer is the distributed simulation engine, which includes an event handler and an interpreter which executes instructions in the code space of a virtual thread.

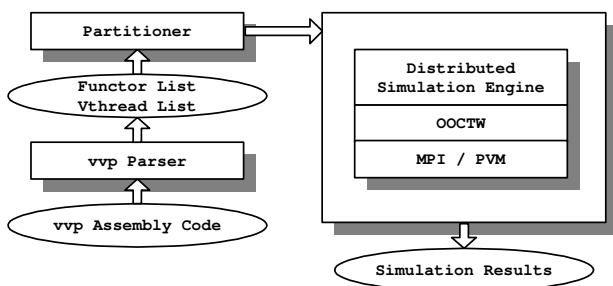


Figure 2. Architecture of DVS

Several partitioning algorithms are included in DVS: RANDOM[3], BFS(Breath-First-Search)[3], DFS(Depth-First-Search)[3] and the design-driven partitioning algorithm.

5 Algorithm and Implementation

In this section, we will explain the implementation of our algorithm in detail.

5.1 hypergraph and data structure

Partitioning algorithms operate on hypergraphs which model a circuit. The gates and wires of the circuit are mapped to the vertices and edges of the hypergraph. In a hypergraph edges may connect two or more vertices and as such it provides a more realistic model of a circuit.

In the circuit hypergraph, we make use of two kinds of vertices. One is an ordinary gate, such as AND, OR, NAND, XOR. The other kind of vertex is a Verilog instance. Actually we can treat it as a super-gate with more complex logic than ordinary gates. We associate the number of gates with each vertex in the hypergraph in order to get an even load distribution. The introduction of super-gates reduces the number of vertices thereby making the algorithm more efficient. This load metric does not work for behavioral Verilog code since we cannot measure the complexity of the behavioral code. This algorithm targets Verilog code at the gate level, i.e. after synthesizing the RTL code.

Figure 3 contains a hypergraph which is composed of two kinds of vertices, gates and super-gates (Verilog instances).

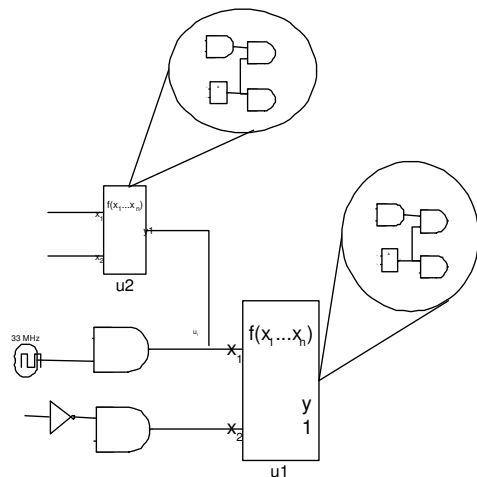


Figure 3. Hypergraph represented by Verilog

In figure 3, there are two Verilog instances, u1 and u2 which are represented by two vertices in the hypergraph. However, in the zoom-out eclipse we see that both u1 and u2 have their own sub-graphs, each of which include multiple gates or Verilog instances.

Before we introduce the data structure used in the algorithm, we define two properties of a vertex. We say that a vertex is not visible if it is inside of a Verilog instance, otherwise it is visible. We say that a vertex is primitive if it cannot be decomposed into multiple vertices, otherwise it is not primitive. Consequently there are four kinds of vertices, as shown in table 1.

Kind	visibility	primitive	example
A	Yes	Yes	Gate outside Vlog instance
B	Yes	No	Top level Vlog instance
C	No	Yes	Gate inside Vlog instance
D	No	No	Sub-level Vlog instance

Table 1. Logic values and their purposes

For example, in figure 3, all of the nodes inside the zoomout ellipse are of kind C, while the node zoomed out is of kind B. The properties of the vertex can change during the partitioning process. For example, the vertex inside of a Verilog instance will become visible after flattening. Any invisible vertex will have the same partition id as its parent. Therefore, only visible vertices will appear in the hypergraph.

The complexity of any partitioning algorithm is proportional to the number of vertices, either $O(n)$ or $O(n*n)$. A reduction in the number of vertices in a hypergraph results in simpler hypergraph and a more efficient partitioning algorithm.

Figure 4 shows the data structure used in the partitioning algorithm. The hypergraph is represented as a vertex vector and an edge vector. Each vertex contains the load, a pointer to its parent, the partition id, the neighbouring vertices list, the Behring edges list and the input ports list. The input ports list contains all of the input ports of the vertex and the internal vertices connected to the input ports while the output ports list contains all of the vertices to which it connects. The ports can be used to flatten a vertex. All of the invisible vertices are expanded into visible vertices when a vertex is flattened. Details of flattening are explained in subsection 5.6.

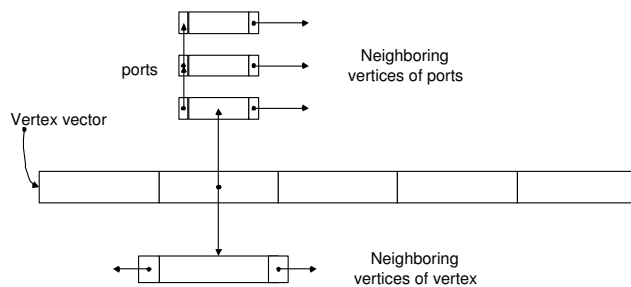


Figure 4. Data structure of the partitioning algorithm

5.2 Verilog parser and hypergraph builder

The Verilog parser reads in the Verilog source code and builds the hypergraph. In the hypergraph, the Verilog in-

stances are treated as super-gates and are therefore represented as one vertex.

5.3 Load balancing constraint

A successful partitioning of a distributed Verilog simulation depends on three factors: communication, load and concurrency. Since it is not possible to optimize each of these factors in isolation from one another, a compromise must be sought. We attempt to minimize the communication between the processors while balancing their computational load.

We define the load on a (processor) node as the number of gates in the partition corresponding to the node. We also define a quantity which measures the relative difference in the load on two (processor) nodes as follows:

$$(load[p1] - load[p2]) / (load[p1] + load[p2]) \quad (1)$$

In the formula 1, load[p1] is the gate number in partition 1 while load[p2] is the gate number in partition 2. The algorithm attempts to balance load by requiring that $(load[p1] - load[p2]) / (load[p1] + load[p2])$ is less than or equal to k .

We have experimented with different values of k , and portray the effect of different choices of k in 6.

5.4 Initial partitioning

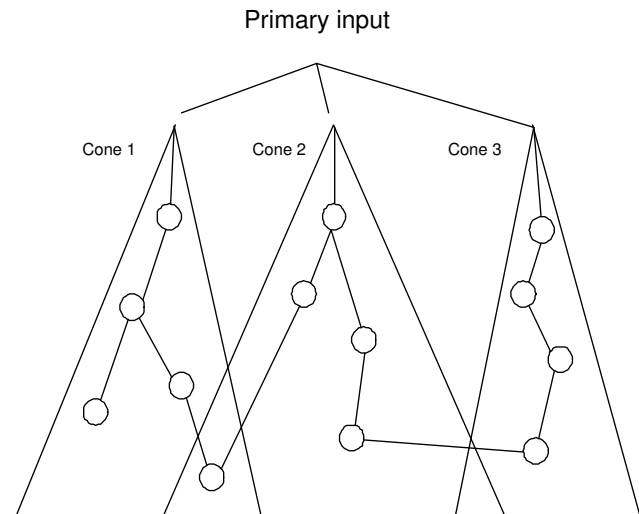


Figure 5. Initial partitioning with cone partitioning algorithm

We use a cone partitioning algorithm [19] as the initial partitioning algorithm. The cones for each primary input are

shown in figure 5. The algorithm traverses the hypergraph from the primary inputs and adds vertices into a partition. If the algorithm detects the vertices that are already added into one partition because of a loop in the circuit, it will traverse from its parent or choose another primary input to continue. The initial partitioning terminates when all of the primary input ports are visited.

The cone partitioning algorithm is known to preserve concurrency in the circuit since it distributes the primary inputs into different partitions.

5.5 Iterative moving

The iterative moving of hypergraph nodes is the same as in the Fiduccia-Mattheyses (FM) [9] algorithm. It modifies the initial partition by a sequence of moves which are organized into passes. At the beginning of a pass, all of the vertices are free to move (they are unlocked), and each possible move is labelled with the immediate change in the total cost which it would cause; this is called the gain of the move (positive gains reduce solution cost, while negative gains increase the cost). The move with the highest gain is executed, and the moved vertex is then locked, i.e. it is not allowed to move again during that pass. Since moving a vertex can change the gains of adjacent vertices, after a move is executed all of the gains of adjacent vertices are updated. The selection and execution of a best-gain move, followed by a gain update, are repeated until every vertex is locked. Then, the best solution seen during the pass is adopted as the starting solution for the next pass. Iterative moving terminates when a pass fails to improve the quality of the solution.

5.6 Flattening

As it turns out, the result obtained using first level super-gates is not always satisfying. For example, if the super-gate is too large, it will destroy the load balance constraint. At this time we need to flatten the super-gate in order to break it into more gates and smaller super-gates. The new hypergraph will be generated after this flattening and the algorithm will continue the iterative moving based on the new hypergraph. The worse case of the algorithm is when all of the super-gates are broken into gates and the hypergraph is exactly same as the hypergraph of the gate-level netlist.

Figure 6 shows the original hypergraph and the result of the flattening. The gates inside the dashed rectangle are flattened from Verilog instance u2.

Currently we choose the super-gate with the maximum gate number in the partitioning. After the flattening, we need to distribute some of the visible nodes from the flattened modules in order to achieve a load balance.

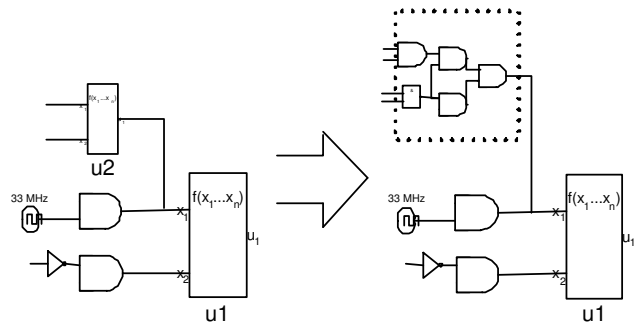


Figure 6. Flattening of the circuit hypergraph

There are two approaches to re-distribute the load after the flattening.

The first is to restart the algorithm from the beginning. After the flattening, a new hypergraph is generated. The algorithm will do the initial partitioning on the new hypergraph, then begin the iterative movement of the hypergraph nodes. It is obvious that this approach will take longer time to finish the partitioning. Hopefully it will generate an improved cutsizes and load balanced partition.

We use the second approach to reduce the partitioning time. This approach redistributes the load between two partitioning based on the previous partitioning result. We define this approach as the incremental load distribution. After partitioning, the lightly loaded partition will pull some nodes from the heavily loaded partition. The pulled nodes are in the cones along the hyperedge between the two partitions. All nodes in the cone are pulled from the heavily loaded partition to the light load partition. The hyperedge which defines the cone is chosen by random.

We observe that cutsizes will increase if we try to achieve a more balanced partition. However, we need to compromise between the cutsizes and load balancing in order to achieve a better simulation speedup. The minimum cutsizes with a load imbalance will trigger a rollback explosion. Details are presented in section 6.

When the iterative moving terminates and the partitioning result satisfies the load balance constraint the partitioning algorithm terminates.

5.7 Putting it all together

Figure 7 contains a flowchart of the algorithm. After the initial cone partitioning, the algorithm will try moving the free vertices between two partitions iteratively until there is no free vertex in the partition. The algorithm then checks whether the load of two partitioning meets the load balancing constraint. If the load balancing constrained is not met, the algorithm will continue to do incremental flattening as discussed in section 5.6. The flattening process and itera-

tive movement process will repeat until minimum cutsizes is achieved and load balancing constraint is met.

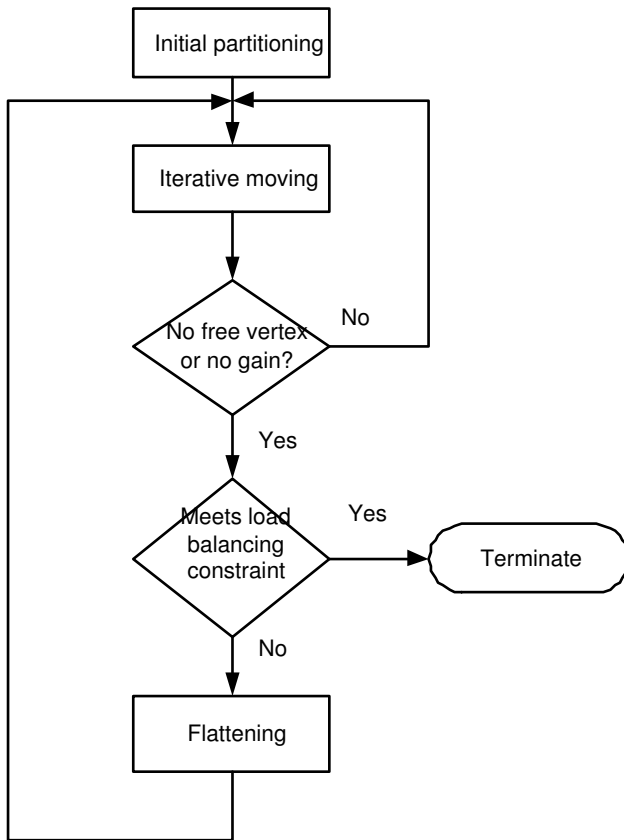


Figure 7. Flowchart of the design-driven partitioning algorithm

6 Experiments

All of our experiments were conducted on a network of 4 computers, each of which has AMD Athlon (CPU 1G) processors and 512M RAM. They are interconnected by a 1Gbit Ethernet network. All of the machines run the Linux operating system while MPICH[10] is used for message passing between different processors. MPICH is a freely available, portable implementation of MPI(Message Passing Interface), a standard for message-passing for distributed-memory applications used in parallel/distributed computing.

We used the synthesized netlist of a Viterbi decoder, which has 388 modules and about 1.2M gates. 1000 random vectors are fed into the circuit. We got the synthesized netlist from Rensselaer Polytechnic Institute. They're conducting the parallel logic simulation on the same circuit[8]. Due to the difficulty of obtaining a large synthesizable in-

k	Flattening	Hyperedge cut	Partitioning time
0.05	209	2428	76.2
0.10	176	1827	54.9
0.15	96	905	35.6
0.20	31	633	31.1
0.25	3	598	25.3
0.30	0	513	21.5

Table 2. cutsizes with design-driven partitioning algorithm

dustry Verilog design, we only conduct experiment on this circuit for now.

We assume a unit gate delay and zero transmission delay on the wires. Each data point collected in the experiments is an average of five simulation runs. The number of machines in the figure doesn't include machine 0, which only contains vthreads[16]. The vthreads generate the events for the simulation. The simulation time for 1 machine is the running time of the DVS without partitioning.

In the experiments, we compare the performance of DVS with the design-driven partitioning algorithm with that of DVS using htmis[13] as the partitioner.

6.1 Cutsizes

We use different values of k to generate different cutsizes. The hyperedge cutsizes is defined as the number of hyperedges that span multiple partitions. Table 2 shows the hyperedge cutsizes produced by our design driven iterative partitioning algorithm while table 3 lists the cutsizes produced by hMetis partitioning algorithm. Parameter k is the load balancing factor defined in formula 1. The Flattening parameter is defined as the number of times flattening is invoked in the partitioning process.

The partitioning time is the average of 5 consecutive executions of the partitioning algorithm. Currently this algorithm is a two way iterative partitioning algorithm. We plan to extend the algorithm to a multi-way partitioning algorithm.

From table 2 and table 3, we know our algorithm reduced cutsizes significantly compared to hMetis partitioning algorithm.

6.2 Simulation time

In our preliminary experiments the simulation took an extremely long time to terminate since DVS consumes a lot of memory and the operating system kept swapping. Swapping makes the performance of DVS even worse than that of a sequential simulation. The reason for this is that DVS

k	Hyperedge cut	Partitioning time(Seconds)
0.05	2675	87
0.10	2673	91
0.15	2673	93
0.20	2669	90
0.25	2668	90
0.30	2665	87

Table 3. cutsize with hmetis partitioning algorithm

k	Cutsize	Simulation time (Seconds)	Speedup
0.05	2428	5834.29	0.62
0.10	1827	3907.90	0.93
0.15	905	2876.13	1.27
0.20	633	2405.87	1.51
0.25	598	2201.98	1.65
0.30	513	2786.54	1.31

Table 4. Simulation time with design-driven partitioning algorithm

treats each gate as an independent LP and each LP needs to save its state, input events and output events. If the GVT is not calculated promptly, the memory overhead for state and event saving can be huge.

In order to attack the problem of memory consumption, we update DVS and only treat the visible nodes in the circuit hypergraph as LPs. For a Verilog module, the states and input events will be saved for each input port while the output events will be saved for each output port. An invisible node without memory inside a Verilog module will not save its state and events. However, the invisible nodes with memory (e.g. a register) will still save their state. If a rollback happens in a Verilog module, every child inside of the Verilog module rolls back along with its parent.

Table 4 shows the simulation times and speedups with different combinations of the load balancing factor and cutsize. The sequential simulation time of the circuit is 3639.70.

From table 4, we know that the minimum cutsize does not always result in the best performance since the performance is also dependent on load balancing. We got the best performance with the combination of a cutsize of 598 and a static load balancing factor of 0.25.

Without a good partitioning algorithm, the distributed simulation is slower than the sequential simulation, as shown in the first two rows in table 4.

7 Conclusion

A partitioning algorithm plays an important role in distributed VLSI simulation. Unfortunately, most partitioning algorithms are very costly and do not always yield a good cut size because they operate on a flattened netlist. Our design-driven partitioning algorithm yields a significant reduction in cutsize compared to such algorithms by taking advantage of hierarchical design information. Moreover, it preserves the locality expressed in Verilog modules and instances. The algorithm produces a 4.5 fold reduction in cutsize compared to the hmetis [13] partitioning algorithm. The reduction in cut size and the preservation of locality lead to a 40% faster execution time on two machines than the sequential simulation.

There are many possible ways to improve the algorithm. It can readily be extended to multiway partitioning by applying the design-driven algorithm to the new partition. Currently the algorithm only works for a structural circuit description. We plan to extend it to RTL and mixed mode (RTL + gate) designs.

An interesting extension of the algorithm would be to make it responsive to changes in processor loads. Currently our load metric is the number of gates, which is not adequate for this task. A first step in this direction is to make use of pre-simulation [7]. Re-partitioning could then be done during the course of the simulation.

Due to the difficulty of obtaining a large synthesizable industry Verilog design, we only conduct experiment on this circuit for now. We are exploring the possibility of conducting more experiments on more industry level Verilog design.

References

- [1] A. B. Kahng, A. E. Caldwell and I. L. Markov. Design and implementation of the fiduccia-mattheyses heuristic for vlsi netlist partitioning. In *Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX)*, Baltimore, pages 177–193, Jan. 1999.
- [2] Herve Avril and Carl Tropper. Scalable clustered time warp and logic simulation. *VLSI design*, 00:1–23, 1998.
- [3] M. Bailey, J. Briner, and R. Chamberlain. Parallel logic simulation of vlsi systems. *ACM Computing Surveys*, 26(03):255–295, Sept. 1994.
- [4] K.-H. Chang, H.-W. Wang, Y.-J. Yeh, and S.-Y. Kuo. Automatic partitioner for distributed parallel logic simulation. In *Modelling, Simulation, and Optimization*, volume 429, Aug 2004.

- [5] Chau-Shen Chen, Ting Ting Hwang, and C. L. Liu. Architecture driven circuit partitioning. In *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, volume 9, pages 383–389, April 2001.
- [6] Jong-Sheng Cherng, Sao-Jie Chen, Chia-Chun Tsai, and Jan-Ming Ho. An efficient two-level partitioning algorithm for vlsi circuits. In *Asia and South Pacific Design Automation Conference 1999 (ASP-DAC'99)*, pages 69–72, 1999.
- [7] Chamberlain R. D. and Henderson C. Evaluating the use of presimulation in vlsi circuit partitioning. In *Proc. 1994 Workshop on Parallel and Distributed Simulation*, pages 139–146. Institute of Electrical and Electronics Engineers, 1994.
- [8] L. Zhu et.al. Parallel logic simulation of million-gate vlsi circuits. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MAS-COTS'05)*, 2005.
- [9] C. Fiduccia and R. Matheyses. A linear-time heuristic for improving network partitions. *ACM/IEEE Design Automation Conference*, pages 175–181, 1982.
- [10] Freeware. *MPICH*. <http://www-unix.mcs.anl.gov/mpi/mpich>.
- [11] Vipin Kumar George Karypis, Rajat Aggarwal and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. In *ACM/IEEE Design Automation Conference*, pages 526–529, 1997.
- [12] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, 1985.
- [13] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Transactions on VLSI Systems*, 7(1):69–79, 1999.
- [14] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.
- [15] H. K. Kim and J. Jean. Concurrency preserving partitioning(cpp) for parallel logic simulation. In *10th Workshop on parallel and distributed simulation(PADS'95)*, pages 98–105, May 1996.
- [16] Lijun Li, Hai Huang, and Carl Tropper. Dvs: an object-oriented framework for distributed verilog simulation. In *Parallel and Distributed Simulation, 2003. (PADS 2003)*, pages 173–180, June 2003.
- [17] Tun Li, Yang Guo, and Si-Kun Li. Design and implementation of a parallel verilog simulator: Pvsim. In *Proceedings of the 17th International Conference on VLSI Design (VLSID'04)*, pages 173–180, 2004.
- [18] R.Chamberlain and C.Henderson. Evaluating the use of pre-simulation in vlsi circuit partitioning. In *PADS94*, pages 139–146, 1994.
- [19] G. Saucier, D. Brasen, and J.P. Hiol. Partitioning with cone structures. *IEEE/ACM International Conference on CAD*, pages 236–239, 1993.
- [20] Wenyong Deng Shantanu Dutt. Cluster-aware iterative improvement techniques for partitioning large vlsi circuits. *ACM Transactions on Design Automation of Electronic Systems(TODAES)*, 7(1):91–121, Jan 2002.
- [21] S. Smith, M. Mercer, and B. Underwood. An analysis of several approaches to circuit partitioning for parallel logic simulation. In *Proc. Int. Conference on Computer Design, IEEE*, pages 664–667, 1987.
- [22] Swaminathan Subramanian, Dhananjai M. Rao, and Philip A. Wilsey. Applying multilevel partitioning to parallel logic simulation. In *Parallel and Distributed Computing Practices*, volume 4, pages 37–59, March 2001.
- [23] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language Fourth Edition*. KLUWER Academic Publisher, 1992.
- [24] Carl Tropper. Parallel Discrete-Event Simulation Applications. *Journal of Parallel and Distributed Computing*, 62:327–335, 2002.