A Multiway Design-driven Partitioning Algorithm for Distributed Verilog Simulation

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

Many partitioning algorithms have been proposed for distributed Very-large-scale integration (VLSI) simulation. Typically, they make use of a gate level netlist and attempt to achieve a minimal cutsize subject to a load balance constraint. The algorithm executes on a hypergraph which represents the netlist. 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.

Keywords: Parallel simulation, VLSI, distributed gate level verilog simulation, partitioning algorithm, load balance cutsize

1. Introduction

Modern Very-large-scale integration (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 [1].

Time Warp [2] 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, obtaining 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 comprising millions of gates. It is widely known that partitioning is an Nondeterministic Polynomial-time hard-complete (NP-complete) problem. The consequence of this is that partitioning algorithms provide heuristic solutions and can be trapped in local minima.

All of the partitioning algorithms to date [3–11] are for distributed/parallel VLSI simulation partition gate level netlists. These algorithms were originally devised for floorplanning and placement, and are computationally very expensive. When used for routing and placement this is acceptable. When they are used as a precursor to a parallel simulation, the goal of which is to speed up a simulation, their cost is not acceptable.

The Application-specific integrated circuit (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 [12]. 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 [13] is briefly described in Section 4. In Sec-

SIMULATION, Vol. 85, Issue 4, April 2009 257–270
© 2009 The Society for Modeling and Simulation International
DOI: 10.1177/0037549709102760
Figures 2, 9 appear in color online: http://sim.sagepub.com

tion 6, 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 [14] partitioning based on netlists is presented in Section 7. The last section contains our conclusions and thoughts about future work.

2. Related Work

A great deal of effort has been devoted to the partitioning of logic circuits [15]. The algorithms are graph based in which the role of the graph is played by a netlist. Not all of this work directly translates to our problem because they were devised for problems in which a high computational complexity can be tolerated e.g. minimizing the area of the circuit layout or minimizing the routing between modules. A number of these approaches have a very high computational complexity (e.g. geometric methods or mathematical programming) and are therefore not suitable because of the emphasis on speed in parallel simulation.

To date, partitioning algorithms for parallel logic simulation also have also been graph based. As previously mentioned, the algorithms are heuristic in nature because the graph partitioning problem is NP complete. The heuristics attempt to minimize the amount of communication between partitions while balancing the computational load on each processor involved in the simulation. We provide a summary of the work done in the domain of parallel simulation.

Simpler heuristics for parallel logic simulation include string partitioning, partitioning by input and output cones and random partitioning. Both string and random partitioning have an O(N) complexity, while cone partitioning has a O(N^2) complexity [3] contains a simple comparison of the concurrency and communication of these algorithms. We therefore conclude that output cones produce the best results.

An important category of algorithms for circuit partitioning is iterative algorithms. Iterative algorithms start from an initial partition and try to improve it. The initial partition is produced by another algorithm, usually by one of the simpler heuristics. Well-known iterative algorithms are CLIP/CDIP [9], Metis/hMetis [14] and F-M [4]. It is worthwhile noting that the CLIP [9] algorithm tries to detect and restore the cluster destroyed by the iterative partitioning algorithm applied to the flattened netlist.

A coarsening phase in a multilevel hypergraph partitioning algorithm is introduced in [14]. 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 [14] produces partitions that are consistently better than other widely used algorithms and is 1–2 orders of magnitude faster than other algorithms. Multilevel partitioning is made use of in [16]. Speedups in the 2.5–3.0 range were obtained for medium size ISCAS85 circuits using Time Warp.

Attempts to try to reduce the size of the hypergraph from the bottom up have been made [9, 14]. They extract clusters from the flattened netlist without reducing the quality of the partitioning of the original netlist. We note that 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 cutsize.

Iterative algorithms work on a 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.

The performance of a number of algorithms using Time Warp in terms of their cutsize and speed up have been compared [17]. Several of the algorithms fell into the simple category i.e. depth and breadth-first search, and an algorithm presented by the authors which they CAKE*. CAKE operates on netlists produced by the Verilog Icarus compiler. These netlists have the property of keeping gates associated with modules adjacent to one another. CAKE slices the netlist into pieces corresponding to modules. Also included in the study are the F-M and Clip algorithms. The author observes that the simpler algorithms produce much larger cutsizes then the iterative algorithms, making them poor candidates for the simulation of large circuits. The CAKE algorithm resulted in excellent performance on smaller circuits as a consequence of preserving the modular structure of the gates.

An algorithm which makes use of both input cone partitioning and iterative improvement is described [18], the purpose of which is to balance the load on the partitions. Speedups of 4.1 using 7 processors on the two largest ISCAS85 benchmark circuits [17] were reported. These circuits have approximately 30 000 gates.

Corolla partitioning [19] finds strongly connected regions in a netlist (corollas), combines them into clusters and then assigns clusters to computer nodes. Using the ISCAS85 benchmark circuits and Time Warp, the authors obtained speedups as high as 8 using 18 processors and 6 using 8 processors. Unfortunately, the authors assume that all of the gates are equally active, an assumption which contradicts the experimental evidence that about 15% of the gates in a circuit are active at any instant in time [20]. The worst-case complexity of this algorithm is O(N^3), where N is the number of gates in the circuit. The authors contend that an average fanout of two reduces this complexity to O(N^2). Either complexity is prohibitively high

* CAKE is a name of the partioning algorithm which is similar to cake cutting.

```
Module m1(p1, p2, p3);
  m2 m2a(......);
  m3 m3a(......);
endmodule

Module m2(p1, p2, p3);
  ......
endmodule

Module m3(p1, p2, p3);
  m4 m4a(......);
  m5 m5a(......);
endmodule

module m4(p1, p2, p3)
  ......
endmodule

module m5(p1, p2, p3);
  ......
endmodule
```

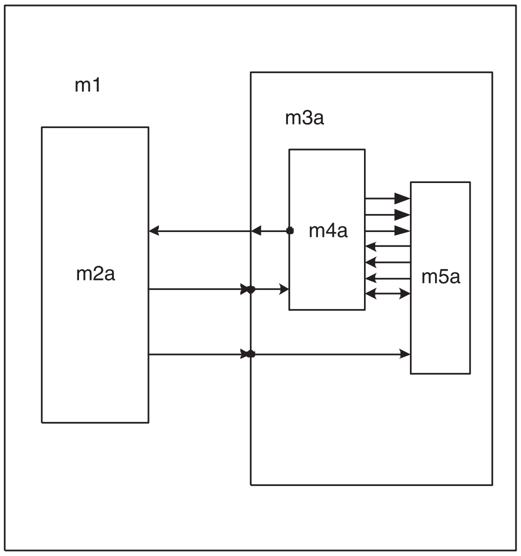

Figure 1. Verilog module/instances and interconnection

for a partitioning algorithm which is intended for simulating multimillion gate circuits.

Concurrency preserving partitioning (CPP) [8] emphasizes the location of concurrency by placing gates which can be concurrently evaluated in different partitions. There are three phases to the algorithm; interprocessor communication and load balancing are taken into account. Speedups in the ISCAS85 benchmarks were computed: a maximum speedup of 30 using 64 processors with pre-simulation on a shared memory machine were observed. Without pre-simulation, speedup results were comparable to those obtained by hMetis. No results were presented about the overhead of the algorithm.

Several algorithms [21–23] operate on netlists and attempt to use information about the circuit to guide the algorithm. These algorithms all have as their principle goal the reduction of cutsize.

Instead of partitioning a circuit prior to the simulation, it is possible to employ dynamic load balancing [24, 25]. The principle idea behind dynamic load balancing is to keep track of a metric which reflects the performance of the simulation and to transfer load between processors when the metric reveals a decrease in performance. [24] uses the number of non-rolled back events as a metric, while [25] uses virtual time progress. Good results were obtained in both of these papers. A decrease in execution time of 25% relative to Time Warp without load balancing on the two largest ISCAS85 circuits was obtained in [24]. While the results obtained by dynamic load balancing were promising for the ISCAS85 circuits, it is not clear how well this approach will scale to large circuits.

In a departure from the approach of graph-based algorithms, [26] describes a bi-partitioning algorithm for Verilog which makes use of its hierarchical design structure. In this paper we extend the algorithms described in [26] to a multiway partitioning algorithm and employ pre-simulation to evaluate the tradeoff between communication and load balancing.

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. The module and its instance are therefore 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 allows a smaller cutsize to be achieved when we partition the circuit. Although Verilog supports cross-module reference, standard design practice discourages such usage.

Figure 1 depicts 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

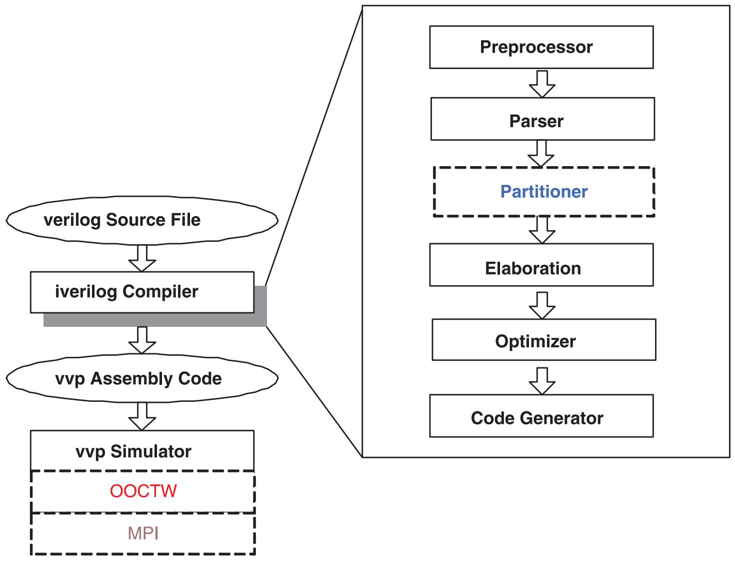

Figure 2. Architecture of DVS

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 cutsize will be smaller and interprocessor communication will be reduced.

We should note that not only does register transfer level (RTL) 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 carried out before elaboration we are able to take advantage of the design information.

4. DVS: A Framework for Distributed Verilog Simulation

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

Figure 2 portrays the architecture of DVS. The three layers of DVS are shown on the right side of the figure. 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 Parallel Virtual Machine (PVM) or Message Passing Interface (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) [27]. 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.

Several partitioning algorithms are included in DVS: RANDOM [28], BFS (breath-first-search) [28], DFS (depth-first-search) [28] and the design-driven partitioning algorithm.

5. Cutsize, Gain and Load-balancing Constraint

A circuit netlist is modeled by a hypergraph $G = \langle V, E\rangle$ where V is the set of vertices while E is the set of nets or wires in the circuit. The edge is not cut if all the vertices of the edge reside in the same partitioning; otherwise the edge is cut. The cost of the cut is defined to be $r-1$ where r is the number of the partitions in which the cut resides. The cutset of the circuit consists of all of the edges which are cut.

5.1 Cutsize and Gain

The cutsize of the circuit is defined to be sum of the cost of all cuts in the cutset as shown in Equation (1). In the formula, c_i represents the ith cut in the circuit while n represents the number of cuts in the circuit.

$$\text{cutsize} = \sum_{i=1}^{n} \text{cost}(c_i). \tag{1}$$

The gain of the vertex movement is defined to be the immediate reduction in cutsize as shown in Equation (2).

In the formula, m is the number of cuts after the vertex movement while n is the number of cuts before the vertex movement. The negative gain means there is no reduction in the cutsize.

$$\text{gain} = \sum_{i=1}^{m} \text{cost}(c_i) - \sum_{i=1}^{n} \text{cost}(c_i). \quad (2)$$

The goal of the iterative movement in the partitioning algorithm is to minimize cutsize through positive gain of the vertex movement.

5.2 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 as the number of gates in the partition assigned to the processor. We make use of a load balancing factor b which allows us to measure the percentage difference in the load on different processors:

$$\text{load} \times (1/k - b/100) <= \text{load}i <= \text{load} \times (1/k + b/100). \quad (3)$$

In Equation (3), load[i] is the number of gates in partition i while load is the number of gates in the circuit. k represents the number of processors involved in the simulation. This load-balancing constraint guarantees that the difference in the load assigned to two different processors is less than $2 \times b$ percent of the total load of the simulation.

We have experimented with different values of k and b, and portray the effect of different choices of b in Section 7.

6. Algorithm and Implementation

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

6.1 Hypergraph and Data Structures

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 therefore providing 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 or XOR. The other kind of vertex is a Verilog instance. We can actually 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 obtain 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.

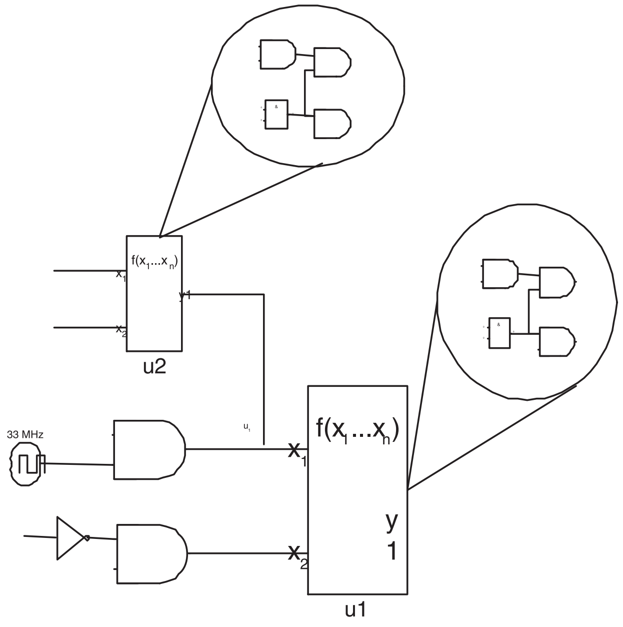

Figure 3. Hypergraph represented by Verilog

Table 1. Logic values and their purposes

Type	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

Figure 3 contains a hypergraph which is composed of two kinds of vertices: gates and super-gates (Verilog instances). There are two Verilog instances in Figure 3: u1 and u2 which are represented by two vertices in the hypergraph. However, in the zoom-out ellipse 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 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.

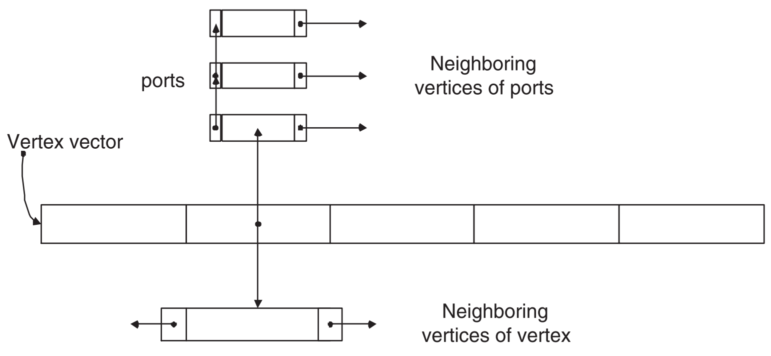

Figure 4. Data structure of the partitioning algorithm

For example, in Figure 3, the zoomout ellipse contains 3 nodes of type C and one node of type B, while the node zoomed out is of type B. The properties of the vertex can change during the partitioning process. For example, the vertex inside 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^2). A reduction in the number of vertices in a hypergraph results in simpler hypergraph and a more efficient partitioning process.

Figure 4 depicts 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 neighboring vertices list, the neighboring 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 Section 6.5.

The bucket is the data structure used to arrange vertices in the order of their gain values. It was first used in [4] in order to improve the runtime performance of the Fiduccia–Mattheyses (FM) [4] algorithm. The bucket data structure is inspired by the bucket sorting algorithm. It has the following two advantages:

1. locating a vertex with the highest gain in the bucket in constant time; and
2. after gain updating, the re-insertion of a vertex into the bucket is accomplished in constant time.

As shown in Figure 5, the bucket is actually a two-dimensional list. All of the vertices on the same row have the same gain value while different rows are ordered by the gain value. All vertices on the same row form a double

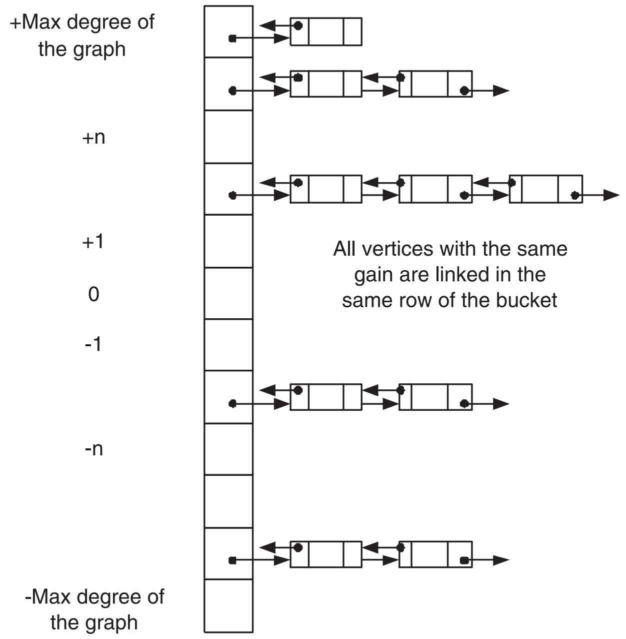
+Max degree of the graph
+n
+1
0
-1
-n
-Max degree of the graph
All vertices with the same gain are linked in the same row of the bucket

Figure 5. Bucket data structure for cell movement

linked list by pointing to the previous and the next vertex. The advantage of double linked list over single linked list is that the remove operation on the double linked list is constant time while the remove operation on single linked list is linear time. It could make a huge difference when the circuit hypergraph has millions of vertices. Our experiments [17] show that the runtime performance of FM [4] based on single linked list could be 300 times slower than the algorithm based on double linked list.

6.2 Verilog Parser and Hypergraph Builder

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

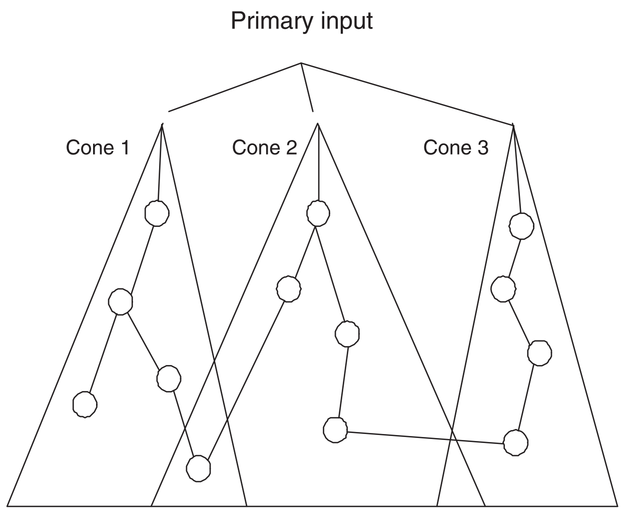

Figure 6. Initial partitioning with cone partitioning algorithm

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

Since we are using cone partitioning [29] for initial partitioning, it is essential that we are able to recognize the primary inputs to the circuit in the Verilog parser. We define an Input port i as a bidirectional port which could be used as input or output port. Currently we define the primary input as an input port and inout port of the Verilog modules.

6.3 Initial Partitioning

We use a cone partitioning algorithm [29] as the initial partitioning algorithm. The cones for each primary input are shown in Figure 6. The algorithm traverses the hypergraph from the primary inputs and adds vertices into a partition. If the algorithm detects the vertices that were already added to a partition because of a loop in the circuit, it will either traverse from its parent or choose another primary input with which to continue. The initial partitioning terminates when all of the primary input ports are visited.

The cone partitioning algorithm preserves the concurrency present in the circuit because it distributes the primary inputs into different partitions.

6.4 Iterative Moving

The iterative moving of hypergraph nodes is the same as in the FM [4] algorithm. The algorithm 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 labeled 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. A detailed explanation of iterative moving follows.

1. Calculate initial gains for all vertices.
2. Insert vertices into buckets of both partitions. After the initial gain calculation of all vertices is finished, all vertices will be inserted into the double linked list at the appropriate bucket location.
3. Locate base vertex from either bucket.
4. Move the selected base vertex.
5. Update gains of the neighboring vertices of the base vertex.
6. Calculate the maximum partial accumulated sum of gains for the current pass.
7. Reverse selected vertices.

6.4.1 Tie-breaking

Tie-breaking strategies play an important role in circuit partitioning because different tie-breaking schemes can lead to different local minima of the cutsize. Our algorithm uses the notion of affinity to break a tie between different cell candidates for a move. Affinity describes how close a vertex is bound to its parent partition. At the beginning of the algorithm, the affinity of a vertex is the number of levels from the root of the hypergraph. This means that the leaf vertices have the smallest affinity with their parent partition.

The idea of affinity extends from the tie-breaking strategy used by CLIP [9], which is to give high priority to those neighbors of the moving cells in the next round of moving based on the locality principle. The CLIP [9] algorithm takes this approach in order to remove large cluster(s) from the cutset. The authors of CLIP observed that large clusters of vertices can still be trapped in the cutset when a small cluster of vertices is extracted out of the cutset.

Sub-clusters which are part of the larger cluster are able to move across the cutline. However, while one sub-cluster moves in one direction, another may move in the opposite direction later on. This movement will finally

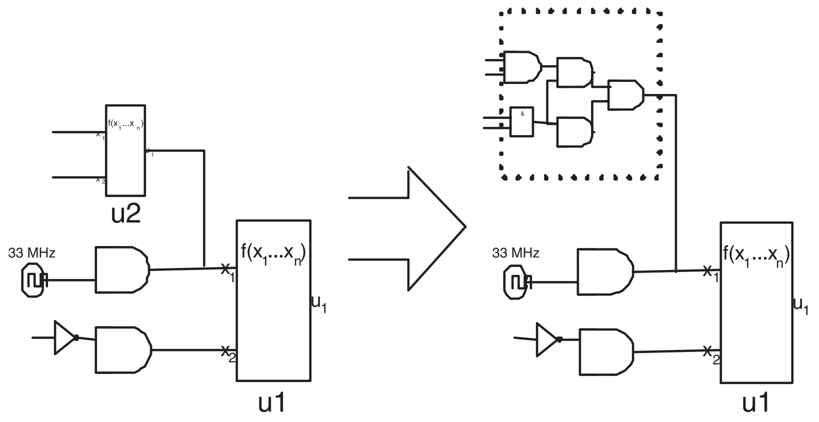

Figure 7. Flattening of the circuit hypergraph

stop when the sub-clusters reside on both sides of the cutline, resulting in the breaking of tightly coupled circuits and introducing a large cutsize. The rationale for the approach is that if one vertex of a large cluster is moved out of the cutset, all of the other vertices from the same cluster should be moved in the same direction in order to eventually move the whole cluster out of the cutset.

6.5 Flattening

The result obtained using first level super-gates is however 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 worst case of the algorithm is when all of the super-gates are broken into gates and the hypergraph is exactly the same as the hypergraph of the gate-level netlist.

Figure 7 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.

There are two approaches to redistribute 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 can substantially increase the partitioning time. However, it could hopefully generate an improved cutsize and load-balanced partition.

The second approach redistributes the load between two partitionings based on the previous partitioning result. We make use of this approach and define it 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 lightly loaded partition. The hyperedge which defines the cone is chosen randomly.

We observe that cutsize may well increase if we try to achieve a more balanced partition. However, we need to compromise between the cutsize and load balancing in order to achieve a better simulation speedup. The minimum cutsize with a load imbalance will actually slow down the simulation, as shown in Section 7.

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

6.6 Pairwise Multiway Iterative Partitioning

The preceding sections have described an algorithm for a two-way partition of the simulation. In this section, we extend the algorithm to more than two machines.

There are two kinds of multiway partitioning algorithms, a flat k-way partitioning [30] and a recursive multiway partitioning algorithm [31–33]. The recursive approach applies bi-partitioning recursively until the desired number of partitions is obtained, while the direct approach forsakes recursion.

The recursive algorithm is computationally simple and fast. However, it suffers from several limitations. In the first place, the number of partitions must be a power of 2. Furthermore, it becomes harder to reduce the cutsize as the algorithm proceeds as the partitioning is performed on successively finer hypergraphs. In effect, global information is lost as we attempt to partition the finer graphs. For these reasons, we have elected to make use of the second approach.

Note that [30] claims that their k-way partitioning algorithm efficiently produces better results.

6.6.1 Pairing of Partitions

We make use of an algorithm which pairs partitions according to the size of the cut-size between pairs of partitions. This is done after obtaining an initial k-way partition via cone partitioning because the algorithm does iterative improvement between these pairs of partitions.

There are a number of ways in which to choose the pairs of partitions.

- Random pairing: It is simple and efficient, but the pairing quality is not good.
- Exhaustive pairing: The algorithm tries all paring combinations. It is computationally complex but produces better results because it can climb out of local minima.
- Cut-based pairing: It pairs the two most tightly or loosely connected partitions. The measurement is the cutsize.
- Gain-based pairing: It pairs two partitions based on the gain in cutsize.

Our algorithm currently employs the cut-based pairing approach. The gain-based pairing approach will be explored in the future.

6.7 Pre-simulation

Pre-simulation is an efficient approach for evaluating the quality of a partition [34]. Evidence is provided that the simulation statistics obtained during the first 10% of the simulation run will not change a great deal during the remainder of the simulation.

We use pre-simulation to evaluate the trade-off between load balance and the communication cost in order to find the best compromise. The criterion used to evaluate a circuit partitioning is simulation speedup during the pre-simulation. The partition which produced the the best speedup for a given number of processors was used as the final partition.

6.8 Putting it all Together

Figure 8 contains a flowchart of the algorithm. After the initial cone partitioning, the pairing process is executed in order to pick candidates for iterative movement. The algorithm then moves free vertices between the two partitions picked by the pairing iteratively until there is no free vertex left or no gain on cutsize could be obtained. The algorithm then checks whether the load meets the load-balancing constraint. If the load-balancing constraint is not met, the algorithm will continue incremental flattening as discussed in Section 6.5. The pairing, iterative movement and flattening process are repeated until no pairing configuration is available. At the end of the partitioning algorithm, the minimum cutsize is achieved and the load-balancing constraint is also met.

7. Experiments

All of our experiments were conducted on a network of 4 computers, each of which has AMD Athlon (1G CPU) processors and 512M RAM. They are connected by a 1Gbyte Ethernet network. All of the machines run the Linux operating system and MPICH [35] is used for message passing between the processors. MPICH is a freely available, portable implementation of message-passing interface (MPI), a standard for message passing for distributed-memory applications used in parallel/distributed computing.

In our experiments we used a synthesized netlist for a Viterbi decoder, which has 388 modules and about 1.2M gates. A million random vectors are fed into the circuit for the full simulation while 10 000 random vectors are used for pre-simulation. We obtained the synthesized netlist from Rensselaer Polytechnic Institute [36]. We restricted ourselves to this circuit as a consequence of the difficulty in obtaining large circuit designs.

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 simulation time for 1 machine is the running time of the DVS with partitioning ready before the simulation. The confidence interval is calculated according to Equation (4) based on 95% confidence level. In Equation (4), σ represents the standard deviation of the samples and n is the number of samples. The distribution of the simulation time samples is assumed to be a normal distribution.

$$\text{ConfidenceInterval} = 1.96 \times (\sigma/\sqrt{n}). \qquad (4)$$

In the experiments, we compare the cutsize produced by the design-driven partitioning algorithm to the cutsize produced by hMetis [14]. The reason for this is that hMetis employs clustering and is therefore appropriate for large circuit hypergraphs. It is also well known for producing partitions with a small number of edges between partitions and has a time complexity of O(N_E) where E is the number of edges in the circuit hypergraph.

7.1 Cutsize

We use different values of k and b to generate different cutsizes. The hyperedge cutsize is defined as the num-

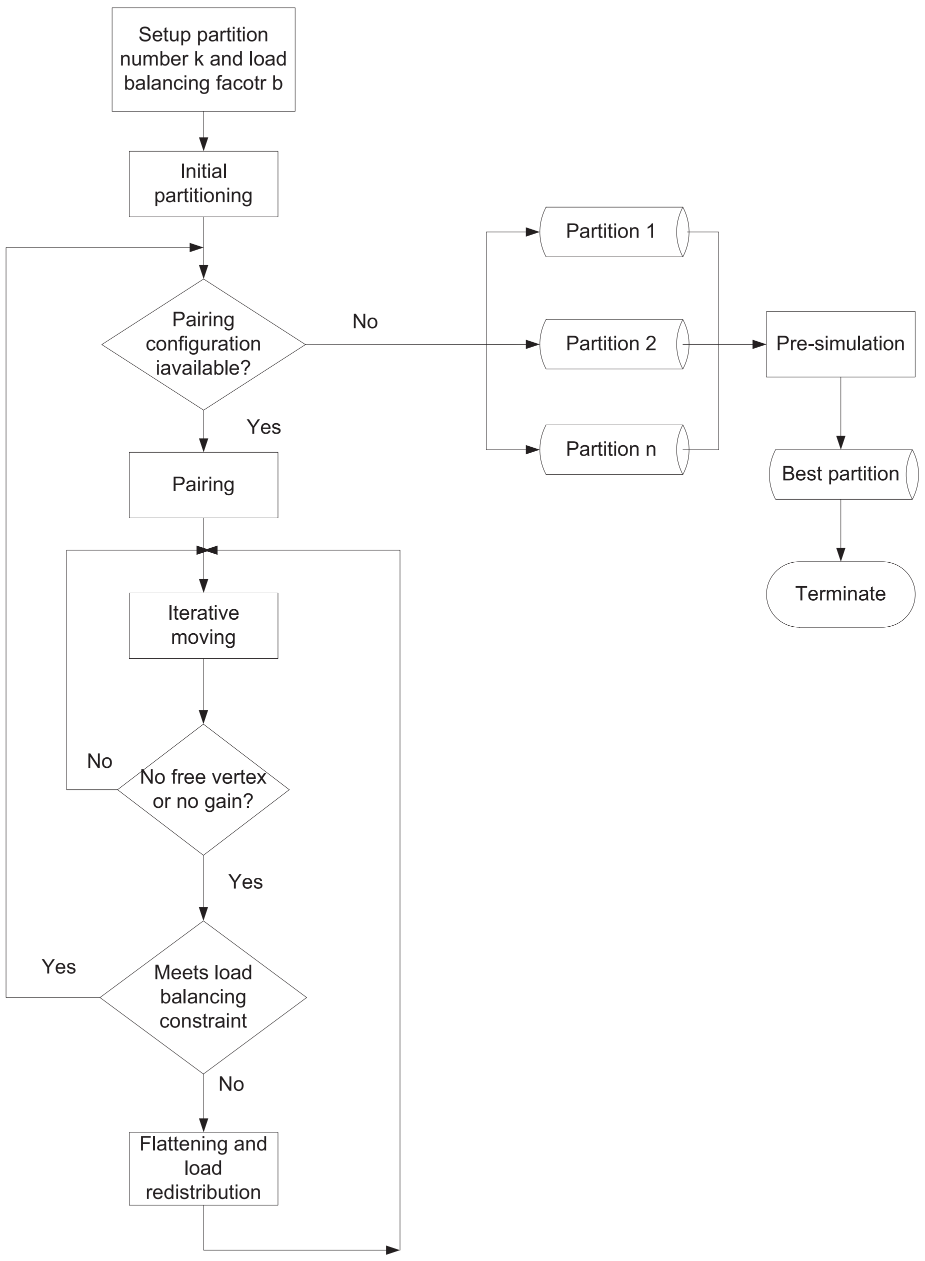
Setup partition number k and load balancing facotr b
Initial partitioning
Pairing configuration iavailable?
No
Yes
Pairing
Iterative moving
No free vertex or no gain?
No
Yes
Meets load balancing constraint
Yes
No
Flattening and load redistribution
Partition 1
Partition 2
Partition n
Pre-simulation
Best partition
Terminate

Figure 8. Flowchart of the design-driven partitioning algorithm

ber of hyperedges that span multiple partitions. In Table 2, Design Driven Partioning (DDP) is the hyperedge cutsize produced by our design-driven iterative partitioning algorithm while hmetis is the cutsize produced by the hMetis partitioning algorithm. The parameter b is the load-balancing factor defined in Equation (3) while k is the number of partitions.

Table 2 reveals that our algorithm resulted in a significantly smaller cutsize than that produced by hMetis.

7.2 Cutsize for ISCAS Benchmark Circuits

The ISCAS89 benchmark circuits have been used extensively to measure the performance of parallel simulation algorithms. We applied our design-driven algorithm to the two largest circuits. Tables 3 and 4 show the cutsize generated by the design-driven partitioning algorithm and the FM [4] partitioning algorithm on ISCAS85 benchmark circuit, s35932 and s38584. The s35932 has 12204 gates, 3861 inverters and 1728 D-type flip-flops. The s38584 has

Table 2. Cutsize comparison between design-driven partitioning and hmetis algorithm

k	*b*	DDP	hmetis
2	2.5	2428	2675
2	5	1827	2673
2	7.5	905	2673
2	10	633	2669
2	12.5	598	2668
2	15	513	2665
3	2.5	2930	2932
3	5	2227	2932
3	7.5	1230	2931
3	10	894	2935
3	12.5	863	2931
3	15	790	2927
4	2.5	3230	3195
4	5	2326	3195
4	7.5	1433	3191
4	10	979	3191
4	12.5	935	3191
4	15	887	3191

Table 3. Cutsize on ISCAS benchmark circuit s39592

k	*b*	DDP	FM
2	2.5	47	47
2	5	47	47
2	7.5	47	47
2	10	46	46
2	12.5	46	46
2	15	46	46
3	2.5	181	181
3	5	181	181
3	7.5	181	181
3	10	181	181
3	12.5	181	181
3	15	181	181
4	2.5	239	239
4	5	239	239
4	7.5	239	239
4	10	231	231
4	12.5	231	231
4	15	231	231

11448 gates, 7805 inverters and 1452 D-type flip-flops. In Tables 3 and 4, DDP is the abbreviation for the design-driven algorithm.

The cutsize produced by the design-driven partitioning algorithm is exactly same as the FM [4] partitioning algorithm. This is expected: flattened netlists contain no design hierarchy and the algorithm degenerates to the FM algorithm.

Table 4. Cutsize on ISCAS benchmark circuit s38584

k	*b*	Cutsize by DDP	Cutsize by FM
2	2.5	53	53
2	5	53	53
2	7.5	53	53
2	10	53	53
2	12.5	52	52
2	15	52	52
3	2.5	167	167
3	5	167	167
3	7.5	167	167
3	10	167	167
3	12.5	167	167
3	15	165	165
4	2.5	211	211
4	5	211	211
4	7.5	211	211
4	10	211	211
4	12.5	211	211
4	15	211	211

7.3 Pre-simulation

We used 10 000 random vectors in our pre-simulation in order to pick the best partition for different combinations of partition number *k* and load-balance factor *b*. The sequential simulation time of the circuit with 10 000 random vectors is 38.93 sec.

Table 5 shows the simulation time and speedup with these combinations. We list the best partitions as determined by the largest speedup in Table 6.

7.4 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 made the performance of DVS even worse than that of a sequential simulation. The reason for this is that DVS 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 is overwhelming.

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 are saved for each input port while the output events are saved for each output port. An invisible node without memory inside a Verilog module will

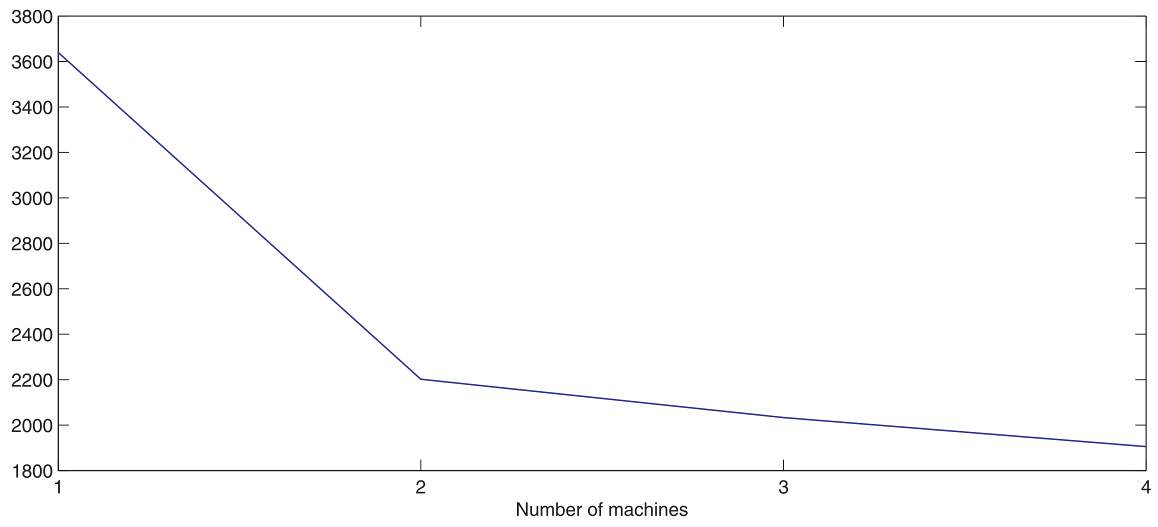

Figure 9. Simulation time

Table 5. Pre-Simulation time with design-driven partitioning algorithm

k	*b*	Cutsize	Sim time	Confidence	Speedup
2	2.5	2428	61.79	2.31	0.62
2	5	1827	41.86	1.91	0.93
2	7.5	905	30.65	1.37	1.27
2	10	633	25.78	1.08	1.51
2	12.5	598	23.59	0.99	1.65
2	15	513	29.72	1.34	1.31
3	2.5	2930	56.42	3.05	0.69
3	5	2227	39.72	2.02	0.98
3	7.5	1230	28.87	1.76	1.35
3	10	894	21.50	1.33	1.81
3	12.5	863	22.37	1.56	1.74
3	15	790	25.44	1.09	1.53
4	2.5	3230	88.47	4.29	0.44
4	5	2326	42.78	2.76	0.91
4	7.5	1433	19.86	0.86	1.96
4	10	979	24.80	0.93	1.57
4	12.5	935	21.04	0.97	1.85
4	15	887	24.18	1.13	1.61

Table 6. Best partition produced by design-driven partitioning algorithm

k	*b*	Cutsize	Sim time	Confidence	Speedup
2	12.5	598	23.59	1.37	1.65
3	10	894	21.50	1.90	1.81
4	7.5	1463	19.86	1.28	1.96

not save its state and events. However, the invisible nodes with memory (e.g. a register) will still save their states. If a rollback happens in a Verilog module, every child inside of the Verilog module rolls back along with its parent.

Table 7. Simulation time with design-driven partitioning algorithm

k	*b*	Cutsize	Sim time	Confidence	Speedup
2	12.5	598	2201.98	53.74	1.65
3	10	894	2033.35	49.88	1.79
4	7.5	1463	1905.60	47.96	1.91

Table 7 and Figure 9 show 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 7, we know that the minimum cutsize does not always result in the best performance since the performance is also dependent on load balancing. We obtained the best performance with the combination of a cutsize of 598 and a static load-balancing factor of 0.25 on two machines. From the data in Table 7, we also observed that the load balancing becomes more and more important as the number of machines increases from 2 to 4. Because of increasing cutsize, we do not see much reduction of the simulation time as the number of machines increases from 2 to 4. This is a consequence of the size of the design. As the number of machines increases, the circuit is divided more finely and more design hierarchy is destroyed. In short, the communication cost offsets the gain from the load distribution.

We also notice that the speedups of the full simulation with 1 million random vectors are slightly less than the speedups achieved from pre-simulation with 10 000 random vectors. We attribute this to the cost of Time Warp. As the simulation runs longer, the overhead costs of Time

Warp (fossil collection and GVT calculation) increase significantly.

Without a good partitioning algorithm, the distributed simulation is slower than the sequential simulation, as shown in the first two rows in Table 7.

8. 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 cutsize 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 [14] partitioning algorithm when applied to the circuit used in our experiments. The reduction in cutsize and the preservation of locality lead to a speedup of 1.91 on four machines compared to the sequential simulation.

An interesting extension of the algorithm would be to make it responsive to changes in processor loads at runtime. Currently our load metric is the number of gates, which is not adequate for this task.

Due to the difficulty of obtaining a large synthesizable industry Verilog design, we have only made use of the circuit described in this paper. We are exploring the possibility of conducting more experiments on large, industrial-quality Verilog designs.

9. References

[1] Tropper, C. 2002. Parallel Discrete-Event Simulation Applications. *Journal of Parallel and Distributed Computing* 62, 327–335.

[2] Jefferson, D. 1985. Virtual time. *ACM Transactions on Programming Lauguages and Systems* 7(3), 405–425.

[3] Smith, S., M. Mercer, and B. Underwood. 1987. An analysis of several approaches to circuit partitioning for parallel logic simulation. *In Proceedings of International Conference on Computer Design, IEEE*, pp. 664–667.

[4] Fiduccia, C. and R. Matheyses. 1982. A linear-time heuristic for improving network partitions. *ACM/IEEE Design Automation Conference* pp. 175–181.

[5] A. E. Caldwell, A. B. K. and I. L. Markov. 1999. Design and implementation of the Fiduccia–Mattheyses heuristic for VLSI netlist partitioning. *In Proceedings of Workshop on Algorithm Engineering and Experimentation (ALENEX), Baltimore*, pp. 177–193.

[6] Chamberlain, R. and C. Henderson. 1994. Evaluating the use of presimulation in VLSI circuit partitioning. *In PADS94*, pp. 139–146.

[7] Subramanian, S., D. M. Rao, and P. A. Wilsey. 2001. Applying multilevel partitioning to parallel logic simulation. *In Parallel and Distributed Computing Practices*, volume 4, pp. 37–59. URL citeseer.nj.nec.com/448559.html.

[8] Kim, H. K. and J. Jean. 1996. Concurrency preserving partitioning(cpp) for parallel logic simulation. *In 10th Workshop on parallel and distributed simulation(PADS'95)*, pp. 98–105.

[9] Shantanu Dutt, W. D. 2002. Cluster-aware iterative improvement techniques for partitioning large VLSI circuits. *ACM Transactions on Design Automation of Electronic Systems(TODAES)* 7(1), 91–121. URL citeseer.nj.nec.com/475548.html.

[10] George Karypis, V. K., Rajat Aggarwal and S. Shekhar. 1997. Multilevel hypergraph partitioning: Applications in VLSI domain. *In ACM/IEEE Design Automation Conference*, pp. 526–529. URL citeseer.nj.nec.com/karypis97multilevel.html.

[11] Karypis, G. and V. Kumar. 1995. 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. URL citeseer.nj.nec.com/karypis98fast.html.

[12] Thomas, D. E. and P. R. Moorby. 1992. *The Verilog Hardware Description Language Fourth Edition*. KLUWER Academic Publisher, Boston, MA, USA.

[13] Li, L., H. Huang, and C. Tropper. 2003. DVS: an object-oriented frame-work for distributed verilog simulation. *In Parallel and Distributed Simulation, 2003. (PADS 2003)*, pp. 173–180.

[14] Karypis, G., R. Aggarwal, V. Kumar, and S. Shekhar. 1999. Multilevel hypergraph partitioning: Applications in VLSI domain. *IEEE Transactions on VLSI Systems* 7(1), 69–79.

[15] Alpert, C. J. and A. B. Kahng. 1995. Recent directions in netlist partitioning: A survey. *Integr. VLSI Journal* 19(1–2), 1–81.

[16] Subramanian, S., D. M. Rao, and P. A. Wilsey. 2001. Applying multilevel partitioning to parallel logic simulation. *In Parallel and Distributed Computing Practices*, volume 4, pp. 37–59. URL citeseer.nj.nec.com/448559.html.

[17] Huang, H. 2003. *A Partitioning Framework for Distributed Verilog Simulation*. Master's thesis, School of Computer Science, McGill University.

[18] Manjikian, N. and W. Loucks. 1993. High performance parallel logic simulation on a network of workstations. *In 7th Workshop on parallel and distributed simulation(PADS'93)*, pp. 85–93.

[19] Sporref, C. and H. Bauer. 1993. Corolla partitioning for distributed logic simulation of VLSI circuits. *In 7th Workshop on parallel and distributed simulation(PADS'93)*, pp. 85–92.

[20] D., C. R. and H. C. 1994. Evaluating the use of presimulation in VLSI circuit partitioning. *In Proceedings of 1994 Workshop on Parallel and Distributed Simulation*, pp. 139–146.

[21] Chen, C.-S., T. T. Hwang, and C. L. Liu. 2001. Architecture driven circuit partitioning. *In IEEE Trancsactions on Very Large Scale Integration (VLSI) Systems*, volume 9, pp. 383–389.

[22] Cherng, J.-S., S.-J. Chen, C.-C. Tsai, and J.-M. Ho. 1999. An efficient two-level partitioning algorithm for VLSI circuits. *In Asia and South Pacific Design Automation Conference 1999 (ASP-DAC'99)*, pp. 69–72.

[23] D. Behrens, K. Harbich, and E. Barke. 1997. Design driven partitioning. *In Asia and South Pacific Design Automation Conference 1997 (ASP-DAC'97)*, pp. 49–55.

[24] Avril, H. and C. Tropper. 1996. The dynamic load balancing of clustered time warp for logic simulations. *In 10th Workshop on parallel and distributed simulation(PADS'96)*, pp. 20–27.

[25] Schlagenhaft, R., M. R. C. Sporrer, and H. Bauer. 1995. Dynamic load balancing of a multi-cluster simulator on a network of workstations. *In Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 175–180.

[26] Li, L. and C. Tropper. 2007. A design-driven iterative partitioning algorithm for distributed verilog simulation. *In 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS 2007)*, pp. 173–180.

[27] Avril, H. and C. Tropper. 1998. Scalable clustered time warp and logic simulation. *VLSI design* 9(03), 291–313.

[28] Bailey, M., J. Briner, and R. Chamberlain. 1994. Parallel logic simulation of VLSI systems. *ACM Computing Surveys* 26(03), 255–295.

[29] Saucier, G., D. Brasen, and J. Hiol. 1993. Partitioning with cone structures. *IEEE/ACM International Conference on CAD* pp. 236–239.

[30] Cong, J. and S. Lim. 1998. multiway partitioning with pairwise movement. *In Proceedings of the IEEE/ACM ICCAD*, pp. 512–516.

[31] Drechsler, R., W. Gnther, T. Eschbach, L. Linhard, and G. Angst. 2002. Recursive bi-partitioning of netlists for large number of partitions. *In Euromicro Symposium on Digital System Design (DSD'02)*, pp. 38–44.

[32] Areibi, S. 2001. Recursive and flat partitioning for VLSI circuit design. *In Proceedings of 13th International Conference on Microelectronics*, pp. 237–240.

[33] Simon, H. D. and S.-H. Teng. 1995. How good is recursive bisection. *SIAM Journal on Scientific Computing* 18(5), 1436–1445.

[34] D., C. R. and H. C. 1994. Evaluating the use of presimulation in VLSI circuit partitioning. *In Proceedings of 1994 Workshop on Parallel and Distributed Simulation*, pp. 139–146.

[35] Freeware. *MPICH*. http://www-unix.mcs.anl.gov/mpi/mpich.

[36] Zhu, L., Chen, G., Szymanski, B. K., Tropper, C., and Tang, T. 2005. Parallel logic simulation of million-gate VLSI circuits. *In 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, pp. 521–524.