PARALLEL LOGIC SIMULATION OF MILLION-GATE VLSI CIRCUITS

By

Lijuan Zhu

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF COMPUTER SCIENCE

Approved:

Boleslaw Szymanski
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2005
(For Graduation August 2005)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I am deeply indebted to my advisor, Prof. Boleslaw Szymanski, for continuous support, patience, encouragement and stimulating suggestions in all the time of my research and writing of this thesis. Without his support and help, this thesis would not be possible.

I thank Prof. Carl Tropper from School of Computer Science, McGill University. I appreciate his good advice and nice cooperation. I thank Prof. Tong Zhang from ECSE, RPI, who providing me the circuits to make experiments with. I also appriciate the fruitful discussion with him and it is a great pleasure to work with him. I thank Prof. Christopher Carothers, who gave a great lecture about parallel simulation, which helped a lot on my research work.

I thank my colleague Gilbert Chen, who really helped me a lot during the past 2 years. I appreciate his invaluable suggestions and experience for me to finish this work. I am very thankful to him that he always can help me out when I get stuck.

I thank my friend Fei, Sun, who helped producing the circuits for my experimental use. I appreciate his help for me to understand the circuit and the Synopsys library.

Finally, I would like to express my sincere gratitude to my parents for their constant support and love.

ABSTRACT

The complexity of today's VLSI chip designs makes verification a step necessary before fabrication. The increasing size of the chips requires very efficient simulation strategies to accelerate the simulation process. As a result, gate-level logic simulation has became an integral component of a VLSI circuit design process that verifies the design and analyzes its behavior. Since the designs constantly grow in size and complexity, there is a need for ever more efficient simulations to keep the gate-level logic verification time acceptably small. The most promising approach is the use of multiple machines to simulate the circuit in parallel, which is referred to parallel logic simulation of circuits. The parallel simulation takes advantage of the concurrency available in the VLSI system to accelerate the simulation task.

Parallel logic simulation has been paid a lot of attention during the past several years, but a high performance simulator is not yet available to VLSI designs. The focus of this thesis is an efficient simulation of large chip designs. We start with a survey of the research done in this field to date, concentrating on parallel logic simulations. Then, we present a design and implementation of a new parallel simulator, called DSIM. Finally, we demonstrate DSIM's efficiency and speed by simulating a large, million gate circuit using different number of processors.

CHAPTER 1
Introduction and background

1.1 VLSI Circuit simulation

The development process of a hardware unit may take several months or even years, and the costs of its fabrication instrumentation may reach several billions of dollars. Therefore circuit simulations done before fabrication have became an important and necessary step to avoid design errors. If undetected, such errors may waste all the time and money invested in the design, because repair of fabricated circuits is currently impractical.

1.1.1 FPGA/ASIC Design Flow

Hardware designs are supported by hardware description languages, or HDLs, such as VHDL (Very High Speed IC Hardware Description Languages) [29] and Verilog [2]. By using a HDL, one can describe arbitrary digital hardware at any level. Chips are designed either in bottom-up or top-down fashion. The preferred style of most Verilog based designs is top-down. Figure 1.1 shows a top-down design and implementation of a FPGA/ASIC unit [34]. HDLs support also different ways to describe the chips. Verilog, for example, provides three levels of abstraction: behavioral level, register-transfer level and gate level.

1.1.2 Four groups of circuit simulation

According to the level of detail, circuit simulation can be classified into four groups [34]:

- Behavioral or functional simulation: Circuit elements are modeled as functional blocks that correspond to the architecture's hardware functional blocks. Functional simulation can take place at the earliest stages of the design. The simulators allow sophisticated data representations and model only the behavior of a design.

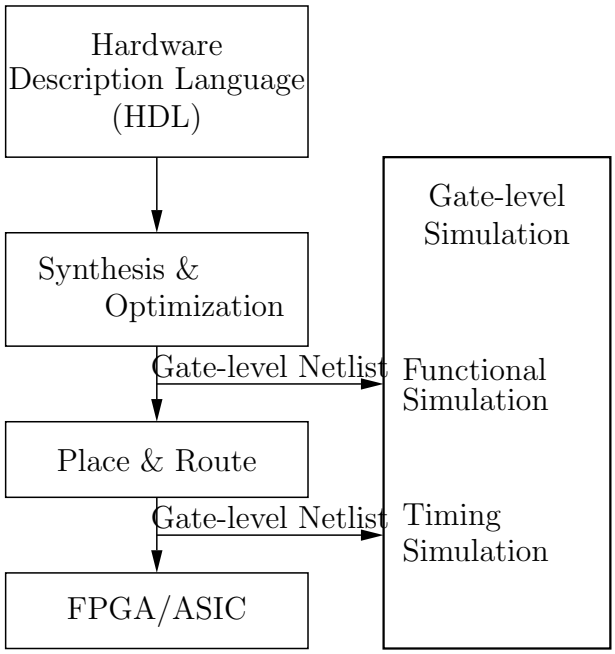

Figure 1.1: FPGA/ASIC Design Flow

- Gate-level logic simulation: Circuit elements are modeled as the collection of logic gates (for example, NAND, OR, D flip-flop) and wires with connectivity information. Figure 1.2 shows a simple example of a gate-level circuit with three logic elements. The input to gate-level simulators is two-valued (0 or 1), and the output is computed based on the truth table modeling the gate. A delay model is associated with each gate, such as zero-delay, unit-delay or multiple-delay models.

- Switch-level simulation: Simulators use the same logic values as gate-level simulators use, but circuit elements are modeled as transistors rather than gates.

- Circuit level simulation: Circuit elements are modeled as transistors, resistors and wires with propagation delays determined by their geometric structure and the underlying technology. These simulators rely on basic physical principles and thus can be highly accurate and general. However, the simulators determine the analog waveforms at nodes of the design, so they are rather slow and unable to process very large design in a reasonable amount of time.

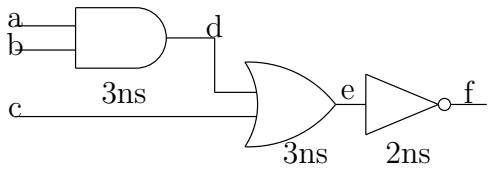

Figure 1.2: A gate-level circuit example

1.1.3 Timing granularity

In logic simulation, there are several possibilities to model the behavior of circuits, each with different timing granularity from very fine-grained timing to coarse-grained timing. Fine-grained timing usually use a time resolution in the range of 0.1ns or smaller, which is more accurate than the coarse-grained timing. Some gate-level simulators may use a single delay for a given element, others may have different days depending on the output or whether the signal is rising or falling. The models can be grouped as the following [43]:

- Continuous time: It is mainly for analog simulations of the lowest level. The currency and voltage is expressed as differential equations in dependency of the time.

- Unit delay: It takes exactly one time unit for the change of a signal to become available.

- Fixed delay: Each element has a constant delay time through the whole simulation time. By this model, the circuit can be simulated more accurately, since different falling and rising times can be simulated.

- Variable delay: This is the most flexible model for the elements. Each element can have variable delays, which is dependent on the output capacity or the state of the simulated system.

1.1.4 Sequential simulation strategies

Gate-level (logic) simulation can be classified into two categories: oblivious and event-driven [34]. In the oblivious simulation, every gate is evaluated once at each simulation cycle, whether or not the inputs have changed. The workload is fixed, and the scheduling can be statically performed at the compile time, thus there is no overhead incurred during the run time. In the event driven simulation, a gate is evaluated only when any of its inputs has changed. For large circuits, event-driven simulation is more efficient because fewer logic gates are evaluated at any time instance. For several reasons, very large and complex systems take substantial amounts of time to simulate even with event-driven simulators [7]. First, increasing the number of simulated gates results in the increase of the number of functional evaluations. Second, the overhead of managing the event queue in the simulator grows with the increase in the number of unprocessed events in the queue. Third, larger the circuit is, larger the number of input vector needed to verify its behavior.

1.2 Special purpose hardware techniques

There are some special purpose hardware techniques used to achieve better performance, such as the Yorktown Simulation Engine (YSE) [46] from IBM and the XP Simulation Booster from Zycad. These hardware accelerator have disadvantages [40]: they are more expensive than general purpose hardware, and the type of elements and delay models that they can handle are very limited. For these reasons, we are not considering hardware accelerators in this paper. Instead, we focus on an event-driven logic simulation for general-purpose computers.

1.3 Parallel Discrete Event-driven Simulation (PDES)

To speed up simulations, parallel discrete event simulation (PDES) for large circuits has been advocated and used. In this approach, the model is composed of some disjoint submodels [43]. The simulation of each submodel is carried by the so-called Logical Process (LP). Each processor takes charge of simulating one or more submodels, or LPs, and each LP interacts with all of the LPs which are influenced by the given LP's local changes. These interactions are accomplished by

messages carrying events between LPs, each event timestamped with the simulation time at which those event should execute. In parallel simulation of circuits, each gate is modeled as an LP, and assigned to a processor. A gate propagates its output signals to the connected gates. If the LP of a connected gate resides on a different processor, the output generates a messages sent to that processor. Each processor maintains an event queue. In order to achieve the correct simulation, it is crucial to ensure that events selected preserve causality.

1.3.1 Event Scheduling

The above requirement is easily satisfied in the sequential simulation by requiring that an event can only produce new events with the time-stamp equal to or greater to its own. This is also natural as the results of the events should not impact the simulated past, but only the simulated present and future. As a result, an event with the smallest time-stamp in the event queue of the sequential simulation is safe to execute (any other event, either already present in the queue or produced in the future must have a time-stamp at least as large as the time-stamp of this event). Even though the same rule about events generating new events applies to each LP in parallel simulation, no longer this rule is sufficient for correctness. LPs exchange messages with events, so there is a danger that a message with the event arriving from some other processor might have a time-stamp smaller than the event at the head of the local queue. Ensuring that such arrival does not invalidate the simulation is the main challenge for the parallel simulation protocols.

1.3.2 Conservative approach

PDES techniques employs two major classes of parallel simulation protocols: conservative and optimistic [25]. A new class of simulation protocol, called lookback, has been recently discovered and presented in [16–18], but the discussion of this protocol is beyond the scope of this thesis as it was not yet applied to circuit simulation.

Under a conservative protocol, an LP executes an event only if it is certain that an event with a smaller time-stamp will not arrive at the LP. LPs with no safe event to process block, which may result in deadlock. Deadlock avoidance and deadlock

detection with recovery are the two ways of dealing with deadlocks in conservative algorithms. We focus on optimistic protocols in this paper. In deadlock avoidance, null messages (a special type of message containing time-stamp but no content) are used to give the lower bounds for the time stamps of the next unprocessed event. This lower bound is used to determine if the event at the head of the event queue is safe to process. This approach reduces the potential of a deadlock, but creates large amount of null messages, thus degrading the performance. The deadlock detection and recovery algorithm eliminates the use of null messages at the cost of deadlock recovery when the deadlock occurs.

1.3.3 Optimistic approach

The original optimistic protocol is known as Time Warp [22, 24]. In Time Warp, an event is processed as soon as it is at the head of the future event queue sorted in the increasing time-stamp. A causality error may occur when a message is received containing an event with a time-stamp which is smaller than the local simulation time. Such a message is called a straggler. After receiving a straggler, an LP recovers by un-doing the effects of processed events with time-stamps larger than that of the straggler. This is accomplished by rolling back the LP to a state which preceding the straggler and by sending negative messages to annihilate events sent to neighboring LPs which have time-stamps larger than that of the straggler. The LP periodically saves its state, so it can restore a previous state when it needs to roll back. In Time Warp, global virtual time (GVT) is defined as the smallest time-stamp among all of the unprocessed positive and negative messages. It is always safe to process an event with a time-stamp equal to or smaller than GVT. Therefore, the events with time-stamp smaller than GVT will never be rolled back, so the memory used to store these events can be reclaimed [25]. Lazy cancellation [26] and aggressive cancellation [25] are two ways of annihilating messages sent out by the rollbacked events. Lazy cancellation cancels those messages only when it is known that they will not be resent again after the rollback, while aggressive cancellation cancels them immediately when the roll back occurs.

1.4 A Viterbi decoder design

As our benchmark, we selected the Viterbi decoder circuits implementing a state-parallel RE Viterbi decoder whose block diagram is shown in Figure 1.3(a). The decoder contains three functional blocks: branch metric unit (BMU) that calculates all the branch metrics; add-compare-select (ACS) units that update the accumulative survivor path metrics; survivor memory unit (SMU) that stores the survivor paths and generate the decoder output. For a trellis with N states, a state-parallel decoder implements all the N ACS units that operate in parallel.

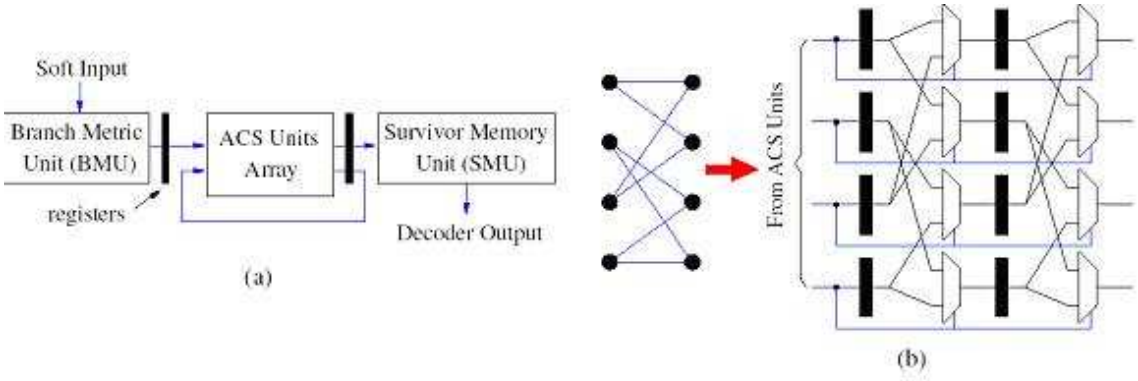

Figure 1.3: (a) The block diagram of state-parallel Viterbi decoder, and (b) example four-state register exchange structure for SMU.

As extensively discussed in the literature (e.g., [9,10,47]), SMU is conventionally designed in two different styles, register exchange (RE) and trace back (TB), targeting different power/complexity versus throughput trade-offs. Basically speaking, RE can easily support very high decoding throughput (e.g., hundreds Mbps or even Gbps) but requires large number of transistors and consumes a lot of power. TB decreases implementation complexity and is quite power-efficient but cannot support very high decoding throughput. In RE Viterbi decoder, as illustrated in Figure 1.3, the decoder output is obtained by simple register shift operation and the critical path typically lies in ACS recursion. On the other hand, in TB Viterbi decoder, certain number of memory accesses are required to obtain each decoder output, which often results in the trace back being the critical path. One important parameter in both RE and TB Viterbi decoders is the decoding decision depth,

which is the length path memory. For convolutional codes, the decision depth selection has been well discussed [3]. What we use is the designed RE Viterbi decoders with the constraint length of 11, corresponding to the number of NAND gates of 1.2 millon.

1.5 Outline of the thesis

The rest of the thesis is organized as follows. The next chapter describes previous research work already done in gate-level circuit simulation. Chapter 3 contains a description of our circuit simulator, introduction of DSIM, and how to build the circuit simulation in DSIM. Chapter 4 provides performance results for our simulations of the Viterbi decoder. Finally, Chapter 5 contains our concluding remarks and plans for future work.

CHAPTER 2
Related work on parallel logic simulation

Many researchers have been developed parallel simulation techniques to speed up logic simulation. Meister gave a good albeit bit dated review of parallel logic simulation in [43].

Briner *et al* [30] implemented a parallel simulator using Time Warp with lazy cancellation. He achieved the speedup of 23 times over sequential simulation on 32 processors of a BBN GP1000 system, running mixed-level simulations. Briner also proposed several improvements over the standard Time Warp to increase the speedup, including incremental state saving, bounding window, and synchronization granularity. In PDES, roll back is achieved by periodically save the entire state of a processor before a next event is processed. The last processed event changes the previous state. In this incremental state saving method, the entire state is saved after every several events processed. All the already processed events are kept in a linked list. When the roll back occurs, the processor restores to the nearest state saved before the straggler. The events between the state and the straggler are re-processed.Bounding window is used to reduce the roll back. Simulation on some processors may get ahead of others, thus receive messages with smaller time-stamp, then roll back needs to be carried out. The bounding window, or time frame is used to prevent the processors from executing events greater than some delta from the GVT to reduce the roll back. That is, The simulation time of a processor can not be advanced exceeding the time frame. However, the bounding window should be carefully chosen. If the bounding window is too large, the time stamps of the events to be processed should be within the bounding window, then the bounding window is useless, any event can be processed as usual. If the bounding window is too small, the processor will block.In synchronization granularity, two synchronization schemes are implemented: inter-processor, and inter-component. In inter-processor synchronization, all LPs in a processor have the same time clock. When roll back occurs, the entire processor needs to be rolled back. In inter-component synchronization,

one LP or several LPs (these LPs can send messages to each other) can have the same time clock. Thus, the straggler event cause the associated LP and the LPs in the same time clock to roll back. This approach can lead to less events rolled back [30].

Bauer *et al* [8] proposed a parallel logic simulator based on event-driven gate-level simulator LDSIM [28]. It achieved speedups between 2 and 4 over the sequential LDSIM simulator on 12 processors for medium sized gates from ISCAS89 benchmarks (sizes of the circuits ranged from 3,500 to 19,200 gates).

Manjikian and Loucks [41] implemented a parallel gate-level simulator on a local area network of workstations. Simulations with large circuits from the ISCAS89 benchmark suite achieved speedups between 2 and 4.2 on 7 processors. The higher speedup of 4.2 was achieved through well-balanced cone partitions [50]. In this partition algorithm, the circuits are regarded as a collection of cones [49]. A circuit can be modeled as a graph. A gate is represented as a node, and a wire connecting two gates is regarded as a link of these two nodes. A cone is formed in the process starting with a primary output of a circuit (the initial set of one node). The nodes with connections to the newly added nodes in the set are added to the set recursively, until the newly added nodes are primary inputs. The partition algorithm is to partition the circuits into blocks with equal number of gates using a depth-first traversal of the circuit to guarantee that the fan-in cone to a primary output is in the same partition as the gate this primary output is in.

Bagrodia *et al* [5] developed a parallel gate-level circuit simulator in the Maisie simulation language [6] and implemented it on both distributed memory and shared memory parallel architectures. They achieved speedup of about 3 on 8 processors of a Sparc1000 for a conservative protocol and about 2 for an optimistic protocol for the four largest circuits (with gates number of 1193, 1667, 2307, and 2418) in the ISCAS85 benchmark suite. The K-FM [11] and K-MAFM [21] partition algorithms were used to partition the circuits. The K-FM algorithm begins with a balanced partition as the initial partition, which is randomly generated. At each step, a gate is moved to another partition, resulting in a currently best partition (has a highest reduction in the cut size, referred as a highest gain. Here, cut size is the number of

links which can be removed to disconnect two partitions.) but no violation of the balance constraint. The algorithm iterates this process until it reaches to a specified number of iterations or no improvement can be made to the partition. K-MAFM is derived from K-FM, with the difference that the given circuits contain no cycles, and they are clustered using the maximum fan-out free cone (MFFC) [31] method before partition.

Meister [44] developed a framework called DVSIM for a parallel event-driven simulator of VLSI designs described in VHDL. Both conservative and optimistic (Time Warp) protocols were implemented. This simulator evolved from the sequential simulator VSIM developed by Levitan [37]. In DVSIM, the four different partitioning algorithms were implemented: round-robin, Kernighan-Lin, K-FM, and soccer partitioning. The round-robin partitioning algorithm assigns gates to the available processors in a circular way. In the acyclic partitioning algorithm, the circuits are represented as directed graph, and partitioned into subgraphs. Each subgraph is mapped to one processor. The Kernighan-Lin the K-FM partitioning algorithms attempt to minimize the number of connections cut by partitions, thus to reduce the communication costs. Their difference between the two is that the Kernighan-Lin algorithm exchanges pairs of gates between two partitions, while the K-FM algorithm moves a gate from one partition to another. The last algorithm discussed in the paper is the soccer partitioning algorithm. It starts by regarding each LP (gate) as a node in a graph. The node with the maximal distance to all other nodes is selected as the first node in one block, then nodes closer to it are added to this block until the number of nodes in this block exceeds a value (this value can be calculated by total number of nodes dividing number of partitions). Then a block is formed, and the nodes in this block are excluded from the graph for later partition. This process is recursively carried out until it reaches to the number of partitions. The author provided simulation results obtained using a conservative protocol on three different sequential circuits with gates 892, 15709, and 40685 from the ISCAS89 benchmark. The results showed that there was no speedup at all for the small circuit. For larger benchmark circuits, the speed up was about 4 on 12 processors. Preliminary results showed that Time Warp protocol with soccer parti-

tioning outperformed the conservative protocol, but Time Warp performed poorly with acyclic partitioning scheme.

Kim [34] implemented a parallel logic simulator on MIMD distributed memory machines. A new partition algorithm, improved Concurrency Preserving Partitioning (iCPP), was proposed. It preserves computation concurrency by assigning gates that can be evaluated at about the same time to the same processor. The iCPP algorithm results in a balance computational load throughout the simulation. Event-lookahead Time Warp (ETW) [35], the hybrid integration of event-lookahead conservative protocol and the Time Warp optimistic protocol was proposed and implemented on an IBM SP2 parallel machine with 10 processors. In the logic simulation of a digital circuits, a gate may be evaluated many times for one primary input vector. For example in Figure 2.1, LP1 schedules four events with different time-stamp to LP2 during the time of one input vector. Actually, the effects of event e1, e2, and e3 are overridden by e4. LP1 can send only e4 to achieve the same effect. That is, e1, e2, and e3 are not necessary. The ETW attempts to look ahead to future events and if possible, to combine multiple events into one with the same effect for the receiving gate. In this example, instead of executing four events, only one event e4 can be executed by LP2. Therefore, this approach speeds up the simulation. The results were compared to the two commercial VHDL sequential simulators: Active VHDL and Quick VHDL simulator. The authors showed that the pure sequential event-driven simulator took about 57-69% of the simulation time of the Active VHDL. The ETW simulation took about 34-95% of the simulation time of the pure sequential simulation. These results were obtained by simulating four circuits with sizes of 2416, 5597, 7951, and 19253 gates. Compared to the Time Warp algorithm, the ETW achieved 20% speed up for a 23843 gate circuit s38417 from the ISCAS89 benchmark.

Lungeanu and Shi [48] developed a parallel compiler and simulator of VHDL designs, achieving almost linear speedup. They proposed a new approach using both a conservative and an optimistic protocols, which they call the dynamic approach. In their dynamic protocol, LPs switch from an optimistic to a conservative protocol if they roll back too much, and vice-versa if they block too much. Simulations were

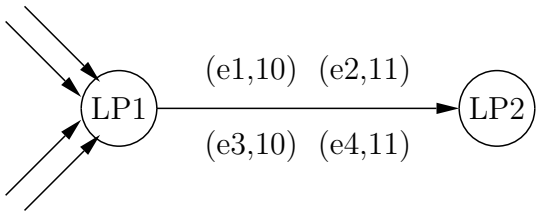

Figure 2.1: Events scheduled for one input vector

carried out on an SGI Challenge parallel machine with 16 processors. The results showed that the speedup was about 11 on a circuit with 14704 gates using dynamic approach.

Williams [51] developed Icarus Verilog, an open-source Electronic Design Automation (EDA) sequential Verilog simulator. Icarus Verilog includes a IVerilog compiler and a Verilog Virtual Processor (VPP), with the VVP assembly code, an intermediate representation of the original circuits. The IVerilog compiler flattens the hierarchical structure of modules, generating a flattened internal netlist. VVP assembly code is the default target format generated from the netlist. The VVP simulator acts as an interpreter of the assembly code. It first parses VVP assembly code to achieve netlist of structural items (inputs, outputs or delay values), then employs the primary input to initialize and drive the simulation.

In [38,39], Li *et al* designed and implemented DVS, an objected-oriented framework for distributed Verilog simulation. The DVS takes the VVP assembly code as input. The VVP parser constructs the structural items, represented by functors and stored in a functor list, which are used by the distributed simulation engine after the circuit partitioning. The distributed simulation engine integrates the original sequential VVP simulator with the Object-oriented Clustered Time Warp (OOCTW) simulator based on the Clustered Time Warp (CTW) [4]. In this algorithm, LPs are grouped into clusters. A sequential algorithm is used within each cluster (that is, events are executed sequentially within a cluster). A Time Warp protocol is used between clusters. The authors conducted experiments on a network of 8 computers simulating a 16bit multiplier with 2416 gates. The results showed that the DVS ran slower than the sequential Icarus Verilog simulator. According to the authors, it

was attributed to the large communication cost, the load imbalance and the small size of the circuits. Large circuits should be simulated by DVS to demonstrate its scalability.

All these parallel logic simulators simulated circuits of quite modest size of about several thousands gates. The simulator described in this thesis has been developed with the explicit goal of simulating large circuits, having millions of gates.

CHAPTER 3
Verilog Simulation

The simulator which we have designed and implemented consists of a translator, a parser and a simulator proper as shown in Figure 3.1.

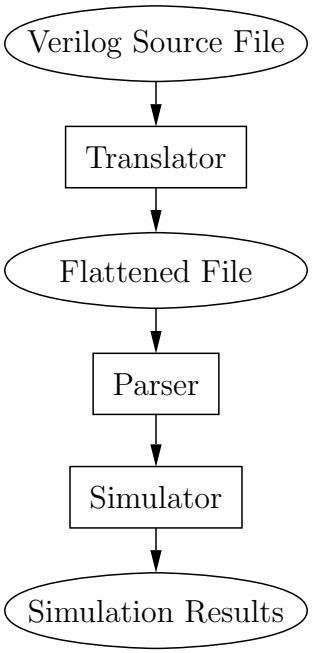

Figure 3.1: The architecture of our simulator

3.1 Translator

Verilog defines modules in a hierarchical structure to enhance the modularity and encapsulation. However, this structure is difficult to process by a simulator. Figure 3.2 shows a module definition for the example in Figure 1.2, and Figure 3.3 shows a hierarchical structure of modules. The goal of the translator is to flatten the hierarchical modules into a netlist without a hierarchical structure, and to generate/output the source file of the netlist with the flattened structure. It is composed of the following components:

- Parsing: The translator first reads in the source file in Verilog format, performing the syntax checking, semantic checking and stores each module in lists of gates, wires, inputs, and outputs.
- Flattening: During the parsing, each time there is a module instantiation, the translator expands the instantiation with the original module definition, renaming all gates and wires.
- Outputting: Using the information stored for the root module (normally, the last module processed), the translator outputs the netlist of this module.

```
module example(a, b, c, f);
input a, b, c;
output f;
wire d, e;
and(.ip1(a), .ip2(b), .op(d));
or(.ip1(d), .ip2(c), .op(e));
inv(.ip(e), .op(f));
endmodule
```

Figure 3.2: Module definition of the example in Figure1.2

```
module t1(...);
   ...
endmodule
module t2(...);
   ...
endmodule
module top(...);
t1 child1(...);
t2 child2(...);
   ...
endmodule
```

Figure 3.3: Module hierarchy

3.2 Parser

The parser mimics the parsing process in the translator, except that its input file is already in the flattened structure. The parser reads in the output from the translator, and then analyzes and stores the gate structures together with the wire connectivity information into the simulator memory.

The reason why we use parsing twice is that after the first parsing process we obtained a file with flattened structure, which is reusable. Had we used only one parsing phrase, the process would have been more complicated, since it would need to do two jobs-deal with the hierarchy, and collect the information about gates. Every time we want to do the simulation, we would need to perform both of these functions. Using two parsing processes to get these two jobs done separately we only need to parse the hierarchical structure once. In this way, we simplify the parsing process and reduce the simulation time.

3.3 DSIM

DSIM outgrew of the project COST [15] that aimed at designing a component oriented discrete event simulation [12–14]. DSIM is a new generation Time Warp simulator developed to support efficient Time Warp simulation on distributed clusters with up to thousands of processors [19]. DSIM features an efficient and scalable GVT (Global Virtual Time) algorithm, referred to as the Time Quantum GVT (TQ-GVT) algorithm, which does not require message acknowledgments, relies on short messages with constant length, and does not use any vector.

The key idea of TQ-GVT is to construct two cuts, as in Mattern's GVT algorithm [42], such that any message sent before the first cut are guaranteed to be received by the time the second cut is completed. However, Mattern's GVT algorithm, or its variants proposed by Mattern and many others ([20, 42, 45]), has to either use multiple rounds to finish the second cut, or, if it can be done within one round, incur waiting time on each processor. In TQ-GVT, the two cuts are dynamically chosen by the GVT master, a processor devoted to running the core of TQ-GVT. Other processors are required to report GVT-relevant information to the GVT master periodically according to a preset interval. After collecting these

reports, the GVT master then constructs the first cut dynamically, by determining the earliest time quantum such that some messages sent from this time quantum are still in transit. The second cut simply consists of the latest reports received from each processor. Thus, the construction of the two cuts always utilizes the latest information available, without using multiple rounds and without incurring extra waiting time, resulting in efficient computation of accurate GVT estimates. TQ-GVT was shown to be able deliver a continuous stream of GVT estimates every 0.1 second even on 1,024 processors [19]. The aggregate network bandwidth consumed by TQ-GVT with such a high update frequency is still less than 1M bytes/second.

In addition to the new GVT algorithm, DSIM uses a modified fossil collection mechanism called Local Fossil Collection, in which fossil collection is done separately by each LP individually, right before an LP attempts to process a new event. Although this technique does not decrease the number of operations, it improves the locality of memory references, since the event memory released in the fossil collection procedure can be immediately reused in the processing of the new event (if there are new events to be scheduled).

DSIM also employs an efficient event management system. For each type of events, it pre-allocates a memory buffer, whose size can be dynamically increased, in order to make constant the complexity of event allocation. To minimize the memory overhead, the event data representing an unprocessed event can share the same memory block with the event data representing the corresponding processed event after the unprocessed event is processed.

DSIM has been demonstrated to simulate a large PHOLD model, consisting of 67,108,862 LPs and 1,073,741,824 events, on 1,024 processors, yielding an event processing rate of 228 million events per second and a speedup of 296. In another study, DSIM has been able to simulate a quarter million spiking neurons, with 50 synaptic connections per neuron, yielding an event processing rate of 351 million events per second and a speedup of 379.

3.4 Building a circuit simulation in DSIM

In our gate-level circuit simulation, gates, primary inputs, and clocks are modeled as individual Logical Processes (LPs). A primary input as well as a clock can be considered as a gate, in which the output replicates the input. Primary inputs to the simulator, are in the form of a list of vector (in hex format, with digits of 0-9 and letters a/A-f/F). Decomposing a vector into bits can produce individual bits for each primary input. Figure 3.4 shows an example of fetching bits from the input vector.

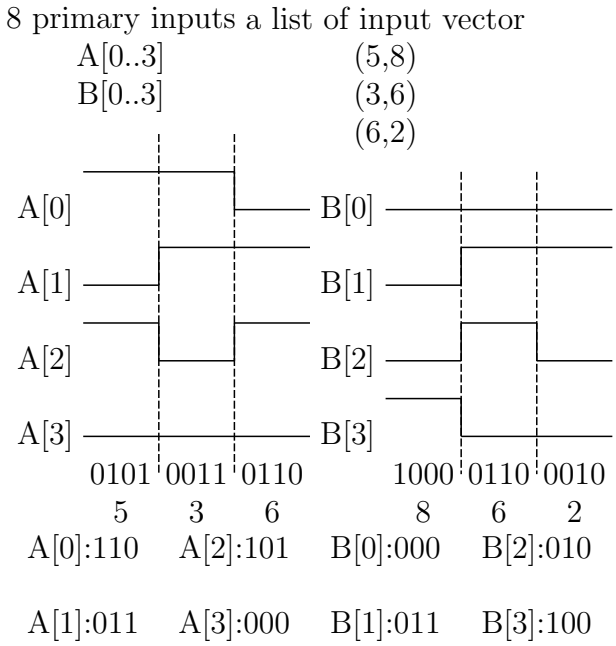

Figure 3.4: Primary inputs

The simulation starts with the LP that models the primary input. It recursively reads a vector from the input list and decomposes it to get the corresponding bits as the input to itself (we model it as a gate replicating its input). The time interval to read the vector is either the time interval of the data supplied, referred to as the data interval or is defined as a parameter of the simulator. The LP that models a clock works similarly to the one that models the primary input. The clock

LP inputs a 0 or 1 bit alternatively every clock interval.

LPs that model gates execute the gate behavior and schedule new events according to their outputs. An event consists of three items: the identifier of the LP to which the event is sent, the bit (0 or 1) representing the output of the gate (LP) sending this event, and the index of the port in the receiving gate (that is the port that is directly connected with the gate sending the event). Each event is also timestamped with the simulation time at which the event should be executed.

At the start of the simulation, an initialization stage activates the primary input LPs that initialize events (with the current simulation time) to its subordinate LPs from the first input vector in the list. They also schedule events destined to themselves with a time-stamp equal to the current simulation time plus the input data interval. The latter events, when executed, will simulate arrival of the next input vector from the input list.

We use an example to illustrate how the LP modelling one primary input fetches input from the list of input vector, and schedules events to subordinate LPs and itself. Suppose the input data interval is 20, A[2] in Figure 3.4 is modelled as LP1 (id=1), and LP10 (id=10), LP20 (id=20) are the subordinate LPs with connection at the second and the first port respectively, which is shown in Figure 3.5.

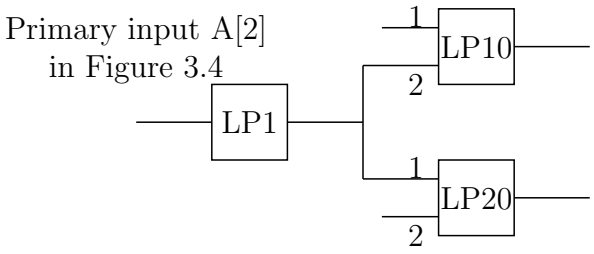

Figure 3.5: An example of wire connection between gates (LPs)

The input list of LP1 is 101 as shown in Figure 3.4. At the start of the simulation, LP1 is fed by bit 1 from the first input. LP1 schedules events (10, <1,2>, 0) and (20, <1,1>, 0) to LP10 and LP20 respectively. It also schedules an event (1, <0,1>, 20) to itself to fetch the second input. In the future, when the simulation time is 20, event (1, <0,1>, 20) will be executed, and new events will be

scheduled to LP10 and LP20 because the output of LP1 will change from 1 to 0. In the meanwhile, a new event (1, <1,1>, 20) for the third input will be sent to itself.

After the initialization stage, the simulator enters the simulation loop. In the body of this loop, first messages from other processors are received, if any and the received events placed in the future event queue. If there are stragglers, the roll back will occur, otherwise the first event at the head of the future event queue is dequeued. The time-stamp of this event becomes the current simulation time and the event is executed, potentially generating new events that are added to the queue. If the current simulated time reaches the predefined total simulation time or there is no more input vector (at the end of the list), the simulation stops. Otherwise, if the time quantum is reached, the TQ-GVT algorithm is invoked. If this is not the case, the the simulation loop body is executed again. The procedure is shown in Figure 3.6.

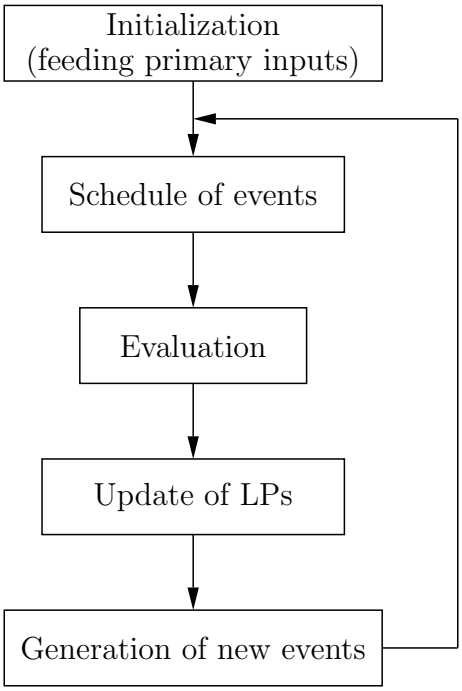

Figure 3.6: Procedure of the simulation

3.5 Circuit partitioning

The placement of circuit elements on the processors can greatly affect the simulation performance. One goal of partitioning is to balance the computation among processors by assuring Before we feed the circuit to the simulator, we need to do the circuit partitioning to distribute the work to each processor as equally as possible. we use a tool called hMeTiS developed at the University of Minnesota [32]. hMeTiS is a tool for partitioning large hypergraphs, especially those in circuit design. The problem is to partition the vertices of a hypergraph into K roughly equal parts such that the hyperedges connecting different parts are minimized. Hyperedge is an extension of an edge by that more than two vertices can be connected by a hyperedge. The hypergraph is such a graph that the edges are replaced by the hyperedges. The algorithms used by hMeTiS are based on multilevel hypergraph partitioning described in [27,33]. By use these algorithsm, hMeTiS has the following advantages: Provides high quality partitions and It is extremely fast.

CHAPTER 4
Simulation experiments and their results

We used the synthesized netlist of the Viterbi decoder obtained through the Synopsys [1] design compiler, which converts a design source code to a netlist file. The simulations were executed on a cluster. Each node of this cluster has 2 800-MHz Intel Pentium III processors with 512 MB memory, connected by a fast Ethernet. The Viterbi decoder circuit that we simulated consists of about 1.2M gates, with 6 primary inputs. The input supplied in our simulation is a list with 1500/500 vectors. The circuit was previously partitioned using hMeTiS shmetis program for 2, 4, 8, 16, 32 parts.

4.1 Experiments and results

There are three factors affecting the simulation time: the total number of events committed, the ratio of the inter-processor events, and the ratio of rollbacks. Table 4.1 summarizes the simulation results of 1500 input vectors, and Table 4.2 summarizes the results of 500 input vectors. Each data collected is the average of 3 consecutive runs.

Table 4.1: Simulation results for the circuit of 1.2M gates with 1500 input vectors

Number of processors	Event processing rate	Speed up	Run time(seconds)	Remote events ratio	Rollbacks ratio
3	137,903	1	1318.744	0.22%	0.14%
5	390,457	2.83	469.192	1.43%	0.19%
9	931,614	6.75	197.57	1.86%	0.23%
17	2,131,558	15.46	86.55	2.34%	0.24%
33	3,839,373	27.84	48.20	4.53%	0.39%

Table 4.2: Simulation results for the circuit of 1.2M gates with 500 input vectors

Number of processors	Event processing rate	Speed up	Run time(seconds)	Remote events ratio	Rollbacks ratio
3	134,061	1	438.805	0.68%	0.43%
5	377,044	2.81	160.60	2.10%	0.59%
9	849,550	6.34	72.274	5.38%	0.65%
17	1,862,908	13.90	33.182	6.40%	0.70%
33	3,033,296	22.63	20.504	11.6%	0.79%

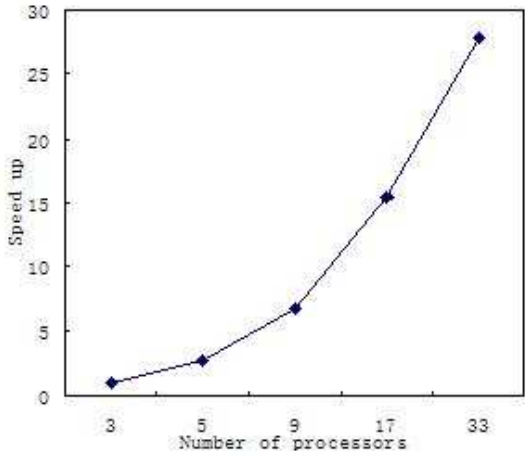

Figure 4.1: Simulation speedup (1500 input vector)

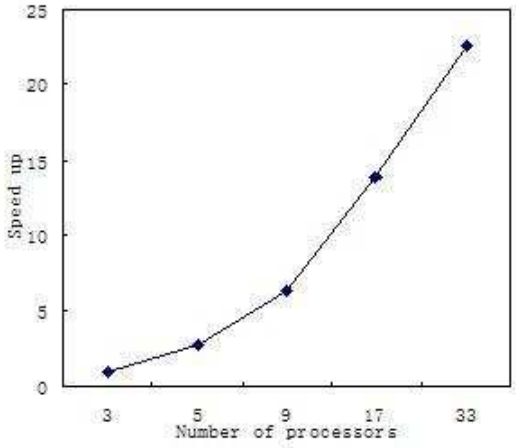

Figure 4.2: Simulation speedup (500 input vector)

4.2 Observation and Analysis

The sequential simulation of this circuit were not done, because none of the cluster nodes had memory sufficient for such a run. However, in parallel simulations, the memory usage is distributed to all of the nodes. Hence a node needs less memory than that in sequential simulation. In DSIM, one processor is used for GVT master, so the results shown in Table 4.1 and Table 4.2 are for 2, 4, 8, 16, and 32 processors. Since the sequential simulation does not complete, we calculate the speedup with 2 processors. From Figure 4.1 and Figure 4.2, we observe the superlinear speedup between 3 and 5 processors, 5 and 9 processors, 9 and 17 processors. These speedups are attributed to less memory needed on a processor because more processors are used. when the available memory is enough for the needed memory on a processor, there is no superlinear speedup between 17 and 33 processors. The speedup between 17 processors and 33 processors is 1.80 for 1500 input vector, and 1.63 for 500 input vectors. However the speedup between 3 processors and 33 processors is as high as 27.84, and 22.63 for 1500 and 500 input vectors respectively. Hence, by increasing 3 processors by the factor of 11, we speed up the computation by the factor of 27 (or 22), a clear sign of a superlinear speedup resulting from improved memory system performance.

The remote event ratios are less than 5% in Table 4.1, and less than 12% in Table 4.2. The rollback ratio is less than 0.4% and 0.8% in Table 4.1 and Table 4.2 respectively. The more remote events, the higher possibility of rollbacks, thus the longer simulation times. From the results, we could infer that the good performance was attributed to the lower remote events ration and rollbacks ratio.

CHAPTER 5
Conclusions and future work

5.1 Conclusions

A parallel logic simulator of a million-gate VLSI circuit has been proposed and implemented using the new simulation engine called DSIM. The circuit experimented is large, with 1.2 Million gates, and the simulation speed is high. Results show that this simulator is capable of efficiently simulating the large circuit with a high speedup. Superlinear speedup is achieved for up to 17 processors. The ratio of speed between 3 processors and 33 processors is about 28.

5.2 Future work

A good partitioning algorithms is central to the success of distributed circuit simulation, as witnessed by our own (and others) experiments. Table 5.1 shows the partitioning times of the circuit we used. The partition time increases a lot when we need more parts. Also, Iterative exchange algorithms such as hMeTiS, used in our experiments, or Clip [23], while effective, can become costly as circuits increase in size. Hence heuristics to decrease their execution time or the use of dynamic load balancing [4] provide important venues for the continued research.

Asynchronous chip simulation is also a very interesting topic. As CMOS is continuously scaling down and the chip is becoming more and more complex, sticking to the conventional synchronous design methodology (the computation and communication within the circuit systems are controlled by a common clock) becomes more and more problematic. As an alternative, asynchronous design methodology (computation and communication are realized by local handshaking) shows some great promise. However, one big problem of using asynchronous circuits is lack of design automation tool including the testing and verification. Asynchronous simulation could be a good extension of the current simulator.

A summary of this thesis has been published at the Proceedings of MASCOT05 [36].

Table 5.1: Partitioning times using hMeTiS (shmetis)

Number of partitions	2	4	8	16	32
Partitioning time(seconds)	232.656	470.236	614.662	783.870	977.857

[1] www.synopsys.com.

[2] Verilog hardware description language standard. IEEE 1364-2001, 2001.

[3] J. B. Anderson and K. Balachandran. Decision depths of convolutional codes. In IEEE Transactions on Information Theory, volume 35, pages 455–459, March 1989.

[4] H. Avril and C. Tropper. Scalable clustered time warp and logic simulation. In *VLSI design*, pages 1–23, 1998.

[5] R. Bagrodia, Y. an Chen, V. Jha, and N. Sonpar. Parallel gate-level circuit simulation on shared memory architectures. In *Computer Aided Design of High Performance Network Wireless Networked Systems*, pages 170–174. NSF, 1995.

[6] R. L. Bagrodia and W.-T. Liao. Maisie: A language for the design of efficient discrete-event simulations. In *IEEE Transactions on Software Engineering*, volume 20, pages 225–238, April 1994.

[7] M. L. Bailey, J. Jack V. Briner, and R. D. Chamberlain. Parallel logic simulation of vlsi systems. In *ACM Computing Surveys*, volume 26, September 1994.

[8] H. Bauer, C. Sporrer, and T. Krodel. On distributed logic simulation using time warp. In *In Proc. VLSI International Conference (IFIP)*, Edinburgh, 1991.

[9] P. J. Black and T. H. Meng. Hybrid survivor path architectures for viterbi decoders. In Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing, pages 433–436, April 1993.

[10] E. Boutillon and N. Demassieux. High speed low power architecture for memory management in a viterbi decoder. In Proc. of IEEE International Symposium on Circuits and Systems, pages 284–287, May 1996.

[11] C.Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *In Proceedings of the ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[12] G. Chen and B. Szymanski. Component-based simulation. In *In Proc. Modeling and Simulation, ESM 2001*, pages 68–75, 2001.

[13] G. Chen and B. Szymanski. Component-oriented simulation architecture: Towards interoperability and interchangeability. In *In Proc. 2001 Winter Simulation Conference*, pages 495–501, 2001.

[14] G. Chen and B. Szymanski. A component model for discrete event simulation,. In *LNCS*, pages 580–594, 2002.

[15] G. Chen and B. Szymanski. Cost: A component-oriented discrete event simulator. In *In Proc. Winter Simulation Conference*, pages 776–782, 2002.

[16] G. Chen and B. Szymanski. Lookahead, rollback and lookback, searching for parallelism in discrete event simulation. In *In Proc. SCSC 2002 Summer Computer Simulation Conference*, 2002.

[17] G. Chen and B. Szymanski. Lookback: A new way of exploiting parallelism in discrete event simulation. In *In Proc. 16th Workshop on Parallel and Distributed Simulation PADS02*, pages 153–162, 2002.

[18] G. Chen and B. Szymanski. Four types of lookback. In *In Proc. 17th Workshop on Parallel and Distributed Simulation*, pages 3–10, 2003.

[19] G. Chen and B. Szymanski. Dsim: Scaling time warp to 1,033. In *Department of Computer Science, Rensselaer Polytechnic Institute*, 2005.

[20] M. Choe and C. Tropper. An efficient gvt computation using snapshots. In *CSMA98*, pages 33–43, 1998.

[21] J. Cong, Z. Li, and R. Bagrodia. Acyclic multiway partitioning of boolean networks. In *In Proceedings of the ACM/IEEE Design Automation Conference*, 1994.

[22] D.Jefferson. Virtual time. In *ACM Trans. Programming Languages and Systems*, volume 7, pages 404–425, July 1985.

[23] S. Dutt and W. Deng. Cluster aware iterative improvement techniques for partitioning large vlsi circuits. In *ACM Trans on Design Automation of Electronic Systems*, pages 91–121, 2002.

[24] R. Fujimoto. Time warp on a shared memory multiprocessor. In *Proc. of the 1989 International Conf. on Parallel Processing*, volume 3, pages 242–249, 1989.

[25] R. M. Fujimoto. Parallel discrete event simulation. In *Communications of the ACM*, volume 33, pages 30–53, 1990.

[26] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *In Proceedings of the SCS Multiconference on Distributed Simulation*, volume 3, pages 61–67, July 1988.

[27] V. K. George Karypis, Rajat Aggarwal and S. Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain.

[28] T. H.Krodel and K. J.Antreich. An accutate model for ambiguity delay simulation. In *Proc. EDAC*, pages 122–127.

[29] IEEE Std. 1076-2002. *IEEE Standard VHDL Language Reference Manual*, 2002 edition.

[30] J. Jack V.Briner, J. L. Ellis, and G. Kedem. Breaking the barrier of parallel simulation of digital systems. In *28th ACM/IEEE Design Automatio Conference*, pages 223–226, 1991.

[31] J.Cong and D. Y. On area/depth tradeoff in lut-based fpga mapping. In *In Proceedings of ACM/IEEE Design Automation Conference*, 1993.

[32] G. Karypis and V. Kumar. Hmetis, a hypergraph partitioning package.

[33] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning.

[34] H. K. Kim. *Parallel Logic Simulation of Digital Circuits*. Phd thesis, Wright State University, 1998.

[35] H. K. Kim and J. Jean. Parallel optimistic logic simulation with event lookahead. In *Proc. of the International Conference on Parallel Processing*, pages 10–15, 1998.

[36] B. S. C. T. L. Zhu, G. Chen and T. Zhang. Parallel logic simulation of million-gate vlsi circuits. In *Mascots*, 2005.

[37] S. Levitan. *Vcomp and Vsim Reference Manual*. University of Pittsburgh, 1993.

[38] L. Li, H. Huang, and C. Tropper. Towards distributed verilog simulation. *I.J. of SIMULATION*, 4(3–4):44–54.

[39] L. Li, H. Huang, and C. Tropper. Dvs: An object-oriented framework for distributed verilog simulation. In *Proceedings of the Seventeenth Workshop on Parallel and Distributed Simulation (PADS'03)*, 2003.

[40] L.Soule. *Parallel Logic Simulation. An Evaluation of Centralized-Time and Distributed-Time Algorithms*. PhD thesis, Standford Univesity, June 1992.

[41] N. Manjikian and W. M. Loucks. High performance parallel logic simulation on a network of workstations. In *Proc. 7th Workshop on Parallel and Distributed Simulation(PADS)*, volume 23, pages 76–84, 1993.

[42] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. In *Journal of Parallel and Distributed Computing*, pages 423–34, 1993.

[43] G. Meister. A survey on parallel logic simulation. Technical report, Department of Computer Science, University of Saarland, 1993.

[44] G. Meister. Evaluation of parallel logic simulation using dvsim. In *HICSS (1)*, pages 397–406, 1996.

[45] K. Perumalla and R. Fujimoto. Virtual time synchronization over unreliable network transport. In *in Proceedings 15th Workshop on Parallel and Distributed Simulation*, page 129, 2001.

[46] G. Pfister. The yorktown simulation engine: Introduction. In *In Preceeding of the 19th ACM/IEEE Design Automation Conference*, pages 170–174, 1982.

[47] C. Rader. Memory management in a Viterbi decoder. In IEEE Transactions on Communications, volume 29, pages 1399–1401, Sept. 1981.

[48] D. L. Richard. Parallel and distributed vhdl simulation.

[49] G. Sauier, D. Brasen, and J. Hiol. Partitioning with cone structures. In *IEEE*, 1993.

[50] 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.

[51] S. Williams. Icarus verilog. Http://icarus.com/eda/verilog.