Optimization techniques for distributed Verilog simulation

Lijun Li

Doctor of Philosophy

School of Computer Science

McGill University

Montreal,Quebec

March 2008

A thesis submitted to McGill University
in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

DEDICATION

To my parents, my dear wife and my lovely daughter.

ACKNOWLEDGEMENTS

I'd like to thank my parents first. My father never forgets to mention my PhD progress whenever he has chance to talk with me over the phone. My mum, who almost spoils me, would punish me if I skipped the school. I still remembered how she dragged me to the school when I tried to avoid an exam in a blizzard since I cannot walk in the snow deep as my knee. But my mum said deep snow cannot be an excuse for absence from the school. My father has been working in the elementary school for over 30 years. He thinks of high education over anything else. In the early 1990s, the money spent on my high school and my university education ate up all of our family savings. I could never pay back the debt I own to my parents.

I need to give my sincere thanks to my supervisor, Carl Tropper. Without his financial support and emotional encouragement, I could never finish my PhD thesis. He even allows me to continue my PhD study part-time after I got a job in AMD. Carl will become my role model in his academic seriousness and creative research methodologies.

I'd like to thank to the members of my PhD committee, who steer my research direction and monitor my research progress. Their constructive feed-back made my PhD dream come true.

Thanks to my friends, David Xu, Jun Wang and Hai Huang. It's really nice to work with them in the distributed simulation lab. The laughter after solving the problems in the lab will become a lifetime happy memory. A special thanks need to give to Hai Huang. He developed a nice partitioning framework, based on which life was much easier for my partitioning algorithm research.

I would like to thank our system staff. Ron always gave me prompt support when I had problem with the machines and Myrinet in lab 107 and the Linux software tools, eg, latex. Diti Anastasopoulos, Lucy St-James and Lise Minogue gave me full support for my PhD study such as organizing my comprehensive exam, proposal and progress report. Lise Minogue even agreed to be my daughter's guarantor when she applied for her Canadian passport.

Thanks to Steve Williams, who developed the Icarus Verilog simulator and made it public as open source code. Thanks to Lijuan Zhu and his supervisor from Rensselaer Polytechnic Institute, who generously lend us Verilog source code with 1 million gates to facilitate our partitioning experiments.

I am so happy that my daughter (Xinyi Li) came into our life during my PhD study. She slowed down my PhD study but she also brought much happiness to my family. She constantly shifted my attention from my computer to her smiling or crying face. Thanks, my little angel. You keep your dad rejuvenated.

The last but not the least, my wife (Xin Ji), deserves my sincere thanks. She endured the poor but busy life of a PhD student without any complains.

Contribution of Authors

The results in Chapter 4 have been published in Parallel and Distributed Simulation(PADS), 2003[1]. The extended result has been published in International Journal of Simulation, Systems, Science & Technology, 2003[2]. My coauthors are Hai Huang and Carl Tropper. Hai Huang designed and implemented a framework for the partitioning of DVS, distributed Verilog simulator. He also implemented the classical FM[3] algorithm and the CLIP[4] algorithm. We are grateful to Carl Tropper for suggesting distributed simulation for Verilog language. The experiment result on ISPD98 benchmark circuit did not appear in the thesis since it was done by Hai Huang.

The results in Chapter 5 has been published in Parallel and Distributed Simulation (PADS) 2004. My coauthor is my supervisor, Carl Tropper.

The results in Chapter 6 has been submitted to SCS SIMULATION journal. The preliminary results has been published in Parallel and Distributed Simulation (PADS) 2007[5]. My coauthor is Carl Tropper. All portions of this thesis that have been published were originally written by myself and carefully corrected and reviewed by coauthors. All algorithms mentioned in the thesis are designed on my own and complemented through discussion with Carl Tropper.

ABSTRACT

Moore's Law states that computational power will roughly double every 18 months. To the semiconductor designer, this means the never-ending challenge of bringing increasingly larger and more complex ICs (Integrated Circuits) to market. It is well known that the principle bottleneck in circuit design is simulation. Uniprocessor simulators may not be able to keep up with increased demands on them for both speed and memory.

This thesis has three main contributions.

The first contribution is a distributed Verilog simulation environment which can be executed on a cluster of workstations using a message-passing library such as MPI (Message Passing Interface). It employs OOCTW as the synchronization backend and takes advantage of the open source code of Icarus Verilog simulator. It is designed to be flexible for future extension and optimization. To our knowledge, DVS is the first distributed Verilog simulator.

The second contribution is event reconstruction, a technique which reduces the overhead caused by event saving. As the name implies, event reconstruction reconstructs input events and anti-events from the differences between adjacent states, and does not save input events in the event queue. Memory consumption and execution time of event reconstruction are compared to the results obtained by dynamic checkpointing revealing that event reconstruction yields a significant reduction in memory utilization and leads to a faster simulation.

The third contribution is a multiway design-driven iterative partitioning algorithm for Verilog based on module instances. We do this in order to take advantage of the design hierarchy information contained in the modules and their instances. A Verilog instance

is represented by one vertex in a circuit hypergraph. The vertex can be flattened into multiple vertices in the event that an adequate 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. The algorithm produces a 4.5 fold reduction in cutsize compared to the hmetis [6] partitioning algorithm. The reduction in cut size and the preservation of locality in the design hierarchy lead to a speedup of 1.91 on four machines compared to the sequential simulation.

ABRÉGÉ

La Loi de Moore stipule que la puissance des processeurs double approximativement tous les 18 mois. Pour le constructeur de semi-conducteurs, cela équivaut à un constant problème d'apporter des CI (Circuits Integrés) de plus en plus larges et complexes sur le marché. Il est bien connu que le goulet d'étranglement dans la conception de circuits réside dans la simulation. Les simulateurs à simple processeur peuvent ne pas suivre les demandes croissantes pour plus de vitesse et de mémoire. Cette thèse présente un environnement de simulation Verilog avec plusieurs techniques d'optimization. Verilog est une langue de conception digitale couramment utilisée. Une simulation distribuée Verilog peut être exécutée sur un groupe de postes de travail en utilisant une librarie passant des messages telle que IPM (Interface Passant des Messages).

Nous décrivons la reconstruction d'événements, une technique qui réduit l'en-tête causé par une sauvegarde d'événements, et comparons sa consommation de mémoire et son temps d'exécution avec les résultats obtenus par checkpointing dynamique. Comme son nom l'indique, la reconstruction d'événements reconstruit la saisie d'événements et d'anti-événements à partir de la difference entre les états adjacents, et ne sauvegarde pas la saisie d'événements dans la queue des événements.

Nous proposons un algorythme partionné redondant à plusieurs voies et orienté vers le design pour Verilog basé sur des instances de modules. Nous faisons cela afin de profiter de l'information hiérarchique de conception contenue dans les modules et leurs instances. Une instance Verilog est représentée par un vertex dans un circuit hypergraphique. Ce vertex peut etre écrasé en plusieurs vertexs dans le cas où une charge adéquate n'est pas produite par une instance basée sur des partitions. Dans ce cas là

l’algorythme écrase la plus grosse instance et déplace les portes entre les partitions afin d’améliorer la charge. Nous présentons nos resultats en utilisant cet algorythme sur un circuit possédant un million de portes décrit sur Verilog.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Moore's Law states that computational power will roughly double every 18 months. To the semiconductor designer, this means a never-ending challenge in bringing increasingly larger and more complex IC(Integrated Circuit) to market.

The complexity and size of digital systems described by Verilog continues to grow. The latest Intel dual-core processor has more than 400 million transistors while the Intel quad-processor has more than 800 million transistors. The introduction of the system-on-chip(SoC), which is intended for use in embedded systems and contains CPUs, memory and analog circuitry on a single chip has only served to exacerbate this problem.

Post-project analysis shows that design and verification account for the majority of the chip development costs. According to a survey of 545 ASIC engineers conducted by EETimes (www.eetimes.com), simulation/verification takes 51% of the design effort on average, as shown in figure 1–1.

Sequential Verilog simulators, or even specialized hardware accelerators, cannot keep up with this pace, and has become a bottleneck of the design process. To accommodate the growing need for increased memory demands as well as the need

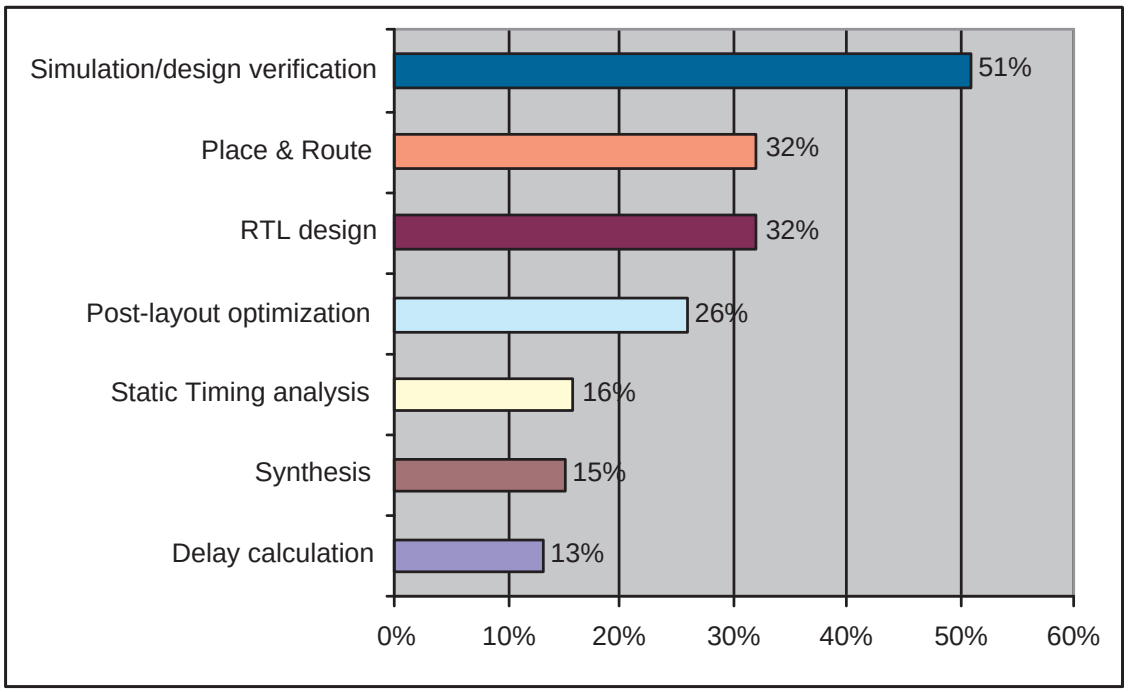


Figure 1–1: Bottleneck in the design cycle

for decreased simulation time, it is necessary to make use of distributed and parallel computer systems[7]. Networks of workstations provide a cost-effective environment for distributed simulation. Time Warp[8] is an appealing technique for parallel and distributed logic simulation of VLSI circuitry because it can potentially uncover higher degrees of parallelism.

Verilog[9] is a widely used language for digital circuit design. This thesis presents a description of our research to date on a distributed Verilog simulation framework and describes the next steps in our research program.

1.1 Motivation of Distributed Verilog Simulation

The rewards for successfully developing a distributed Verilog simulator are substantial. Distributed simulation gives us the ability to simulate much larger circuits than is now possible on one workstation, and to do so in a cost-effective manner if we make use of a cluster of workstations as a simulation platform. We will also have the ability to execute simulations much faster than is possible at present, thereby decreasing the time to design a circuit. Moreover, a distributed Verilog simulation is able to overcome the memory bottleneck for very large logic simulations. With the advent of SoCs and the ever increasing number of transistors which can be packed on a chip, distributed simulation can make an important contribution to VLSI design automation. These contributions include:

- Reduced simulation time

  By dividing a large simulation computation into many sub-computations that can execute concurrently one can reduce the execution time by up to a factor equal to the number of processors that are used. This may be important simply because the simulation takes a long time to execute, e.g., the simulation of millions gates could take days to finish the simulation in order to verify the correctness of the logic design.

- Overcome the physical limit of memory for 32bit computers

  As the ASIC design becomes more complex, the memory requirement for gate-level simulation can easily exceeds the 4G bytes limits of 32bit computers and make the huge investment on the 32bit computing farms in ASIC design industry meaningless. Distributed simulation could take advantage of these 32bit machines

and divide a whole design simulation into several partitions, each of which could be able to run in 4G memory limit.

1.2 Objectives of Distributed Verilog Simulation

We have four major objectives for our research in distributed Verilog simulation.

1. To construct a flexible platform which can make use of open source simulators and allow the addition of new algorithms for distributed Verilog simulation.
2. To investigate significant issues in distributed Verilog simulation.
3. To develop optimization techniques for optimistic simulations, e.g. memory consumption optimization.
4. To develop appropriate partitioning algorithms.

1.3 Overview of the Thesis

In chapter 2, we briefly introduce logic simulation, circuit simulation, hardware design methodologies, hardware design description language and the discrete simulation algorithm employed in the Verilog language.

The chapter 3 is devoted to the introduction of PDES (Parallel Discrete Event Simulation). The two major categories of synchronization algorithm are mentioned, conservative algorithm and optimistic algorithm, also known as Time Warp, are described. The state of art of the Time Warp optimization algorithm is described at the end of this chapter.

The chapter 4 contains the description of DVS, the Distributed Verilog Simulator which we developed in Distributed Simulation Lab of McGill University. The detailed implementation of DVS is explained in this chapter and the preliminary experiment

results are also presented. From the preliminary experiment result, we locate the significant issues inside the simulator and proposes our optimization techniques in the next chapters.

The chapter 5 proposes event reconstruction as a memory reduction techniques for DVS. Historically, most of the memory reduction techniques targets memory reduction for the state queue. Our event reconstruction technique targets event queue. This decision is based on our preliminary experiment result which revealed that the event queue actually consumes more memory than the state queue in distributed Verilog simulation.

The chapter 6 proposes a multi-way design driven iterative partitioning algorithms for distributed Verilog simulation, which could obtain a simulation speedup of 1.91 on 4 machines by taking advantages of design hierarchy information contained in the modules and their instances. A Verilog instance is represented by one vertex in the circuit hypergraph, which could be flattened into multiple vertices in the event that an adequate load balance is not achieved. In this case the partitioning algorithm flattens the largest vertex and move vertices between the partitions to improve the load balance.

The last chapter 7 is the conclusion of the thesis along with several suggestions for further research.

CHAPTER 2

Background

2.1 A brief history of the semiconductor electronic design automation

We give a brief introduction of the ASIC design history in the thesis. The readers should refer to [10] for detail information.

In the infancy of integrated circuit design in the 1960s, traditional prototyping and bread-boarding is dominant while software simulation gradually became accepted by the designers of integrated circuits.

In the 1970s and 1980s, the ASIC design industry gradually adopts standard cells as the building block of the integrated circuits. This allows the designers to design their chips in much shorter time periods.

In the 1990s, the logic synthesis tool became the milestone of the ASIC design history. For the first time, the logic synthesis tool abstracts the integrated circuits to higher level and hides the physics of the circuits. Thus, it reduces the design time significantly. The ASIC circuits and its application increases exponentially in this time.

In the 2000s, the ASIC design challenge is the even increasing complexities characterized by Moore's Law: integrated circuits complexity doubles approximately

every 18 months. This imposes difficulty for the verification of the ASIC design since the simulation for the ASIC circuit becomes the bottleneck of the ASIC design.

2.2 Modern ASIC design flow

Figure 2–1 shows the steps in the modern ASIC design flow.

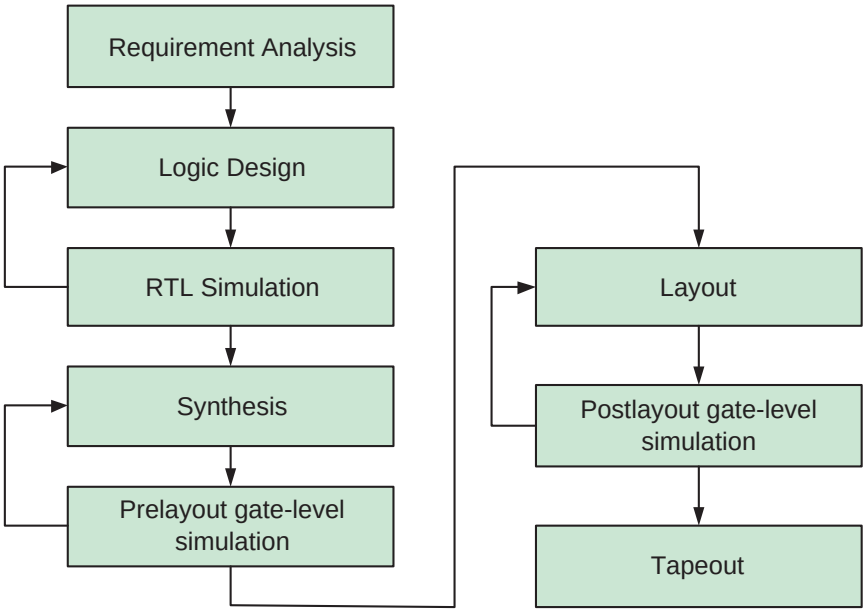


Figure 2–1: ASIC design flowchart

- Requirement analysis

  The ASIC design starts with understanding of the required functions of the ASIC.

- Logic design

  The design engineer constructs a description of an ASIC using a hardware description language such as VHDL[11] or Verilog[9]. This process is analogous to writing a computer program in a high-level programming language. This is usually called the RTL (Register transfer level) design.

- RTL simulation

  Functional correctness is verified by simulation. The common way to verify logical correctness is to feed the input vectors to the digital circuit and compare the simulation result with the expected golden data. If the simulation result failed to match the golden data, we should know that something is wrong with the digital circuit design. Golden data is usually generated by the emulation program written in another programming language such as C language.

- Synthesis

  A logic synthesis tool, such as Design Compiler[12], synthesizes the RTL design into a netlist of standard cells, such as 2 input NOR gate, 2 input NAND gate, inverters, etc.

- Pre-layout gate-level simulation

  The pre-layout gate-level simulation[13] is to the verify the correctness of gate-level netlist generated by the synthesis tool.

- Layout

  The gate-level netlist is processed by a placement tool which places the standard cells onto a region representing the final ASIC. It attempts to find a placement of the standard cells based on the specified constraints such as area of the chip, wire length between blocks, etc.

  The routing tool takes the physical placement of the standard cells and uses the netlist to create the electrical connections between them.

  The final output of place and routing is a set of photo-masks enabling a semiconductor fabrication to produce physical ICs.

Figure 2–2 shows photograph of Power4 processor by IBM after place and route, courtesy of IBM from http://www.research.ibm.com/journal/rd/461/warno1.jpg.

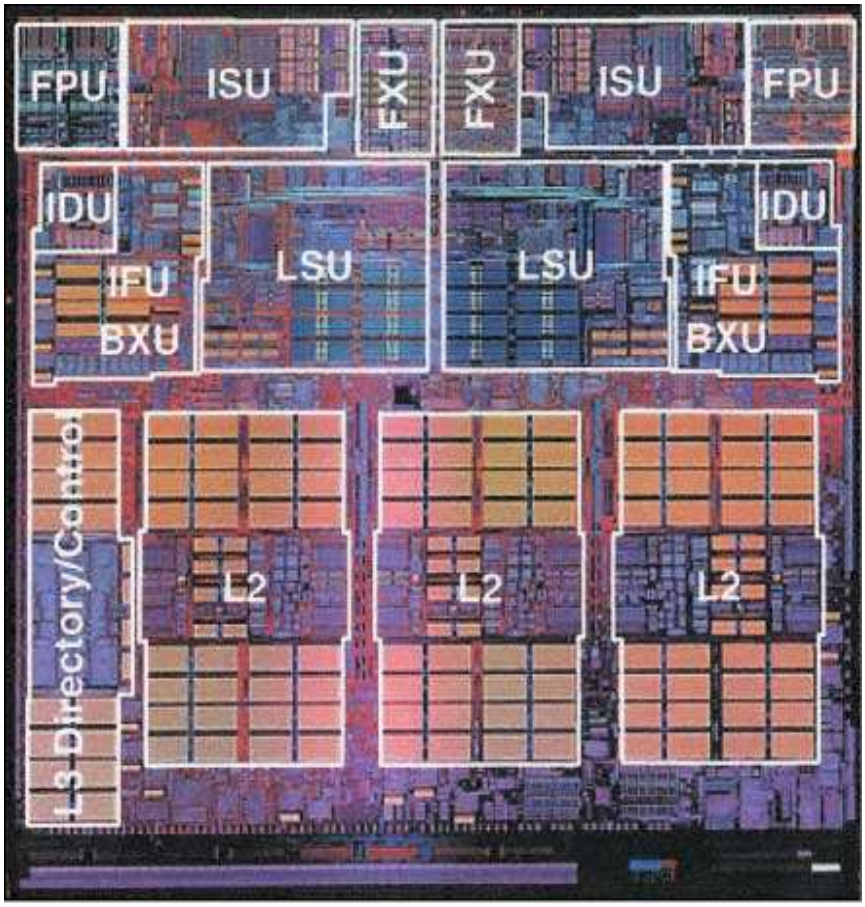


Figure 1

POWER4 chip photograph showing the principal functional units in the microprocessor core and in the memory subsystem.

Figure 2–2: Photograph of Power4 processor by IBM

- Post-layout gate-level simulation

After the layout, the netlist is verified by running simulation again to get rid of the potential interconnection or timing problem introduced by layout.

2.3 The hardware description language

”In electronics, a hardware description language or HDL is any language from a class of computer languages for formal description of electronic circuits. It can describe the circuit’s operation, its design and organization, and tests to verify its operation by means of simulation.”[14]

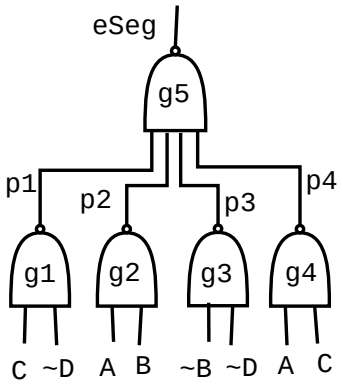
Contrary to the software programming language, an HDL includes syntax elements to express time, concurrency and connectivity which are the primary attributes of hardware.

A simulator is used to simulate the hardware behaviour described by the hardware description language. The simulator could employ either continuous simulation to simulate the analog circuit or discrete event simulation to simulate the digital circuit, as discussed in the section 2.4.

The two most widely-used hardware description languages are VHDL[11] and Verilog[9]. Since the thesis is about distributed Verilog simulation so we only focus on the introduction of Verilog language. The interested readers could read [11] about VHDL language for detail.

2.3.1 Verilog

The Verilog Hardware Description Language is standardized in IEEE standard #1364-1995. It supports both a behavioral description and a structural description of a




```
//structural description
Module binaryToESeg;
   wire eSeg, p1, p2, p3, p4;
   reg A, B, C, D;

   nand #1
     g1(p1, C, ~D),
     g2(p2, A, B),
     g3(p3, ~b, ~D),
     g4(p4, A, C),
     g5(eSeg, p1, p2, p3, p4);
endmodule

//behavioral description
Module binaryToESeg;
   wire eSeg, p1, p2, p3, p4;
   reg A, B, C, D;

   always @(A or B or C or D)
   begin
     eSeg = 1;
     if(~A&D) eSeg = 0;
     if (~A&B&~C) eSeg = 0;
     if (~B&~C&D) eSeg = 0;
   end
endmodule
```

Figure 2–3: Structural and behavioral description of Verilog

digital system. Figure 2–3 shows an example of how Verilog describes an IC design[9]. The figure contains part of a binary to seven segment display driver.

The structural description shows the explicit structure of the circuit and contains all logic gates used and their interconnections. The behavioral description describes the input and output behavior of the circuit. Through the logic synthesis tool, both structural description and behavioral description could produce the same circuit.
In Figure 2–3, the right top shows the structural description of the binary to seven segment display driver circuit while the right bottom shows the behavioral description.

A behavioral description of a flip-flop is shown in figure 2–4.

Verilog describes a digital system as a set of modules. Each module has an interface to other modules (referred to as port(s)) and represents a logical unit in a

```
module tff(input t, input clk, output q);
  always @(posedge clk)
  begin
    if (t==1'b1) q<=~q;
    else q<=q;
  end
endmodule
```

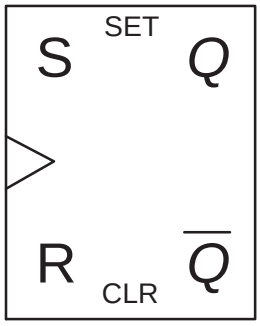


Figure 2–4: Behavioral description of a flip-flop in Verilog

structural description or in a behavioral description. The modules are typically arranged in a hierarchical manner. The hierarchy can be made use of in partitioning in an effort to minimize inter-processor communication.

Verilog is a concurrent language. A digital system can be conceived of as a set of concurrent processes contained in initial blocks, always blocks and continuous assignments. Wait and event control statements can be used to synchronize the concurrent processes. The existence of concurrent processes in Verilog makes it suitable for distributed simulation[15]. A comprehensive description of Verilog can be found in [9].

A Verilog design consists of a hierarchy of modules. Modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. Concurrent and sequential statements define the behaviour of the module by defining the relationships between the ports, wires, and registers. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But all concurrent statements and all begin/end blocks in the design are executed in parallel. A module can also contain one or more instances of another module to define sub-behavior.

Hierarchy is an important feature of Verilog. With the use of design hierarchy information, the partitioning algorithm do not have to go deep into the lowest level of the circuit if the load balance constraint could be met. Moreover, because of the encapsulation property of the module, the circuit graph is simplified so the partitioning efficiency could be improved. Our multiway design driven partitioning algorithm takes advantage of the module and hierarchy in Verilog and yields a significant reduction in cutsize compared to other partitioning algorithms working on the pure flat netlist. Details of the hierarchy feature of Verilog are described in section 6.5.

A subset of statements in the language is synthesizable. If the modules in a design contain only synthesizable statements, the logic synthesis tool can be used to synthesize the design into a netlist that describes the the logic gates and their connections. The netlist may then be transformed into the photo-mask for the final fabrication.

2.4 Continuous and discrete event simulation

”Simulation[16] is the representation of operations and attributes of one system through the medium of another. The attribute set of the simulation model at any given instant is referred to as the simulation state. The simulation state actually consists of all the states at a particular time.”

There are two main categories of simulation: discrete simulation[17] and continuous simulation[18].

In a continuous simulation[18], the simulation state changes continuously with simulation time. The simulation state is characterized by differential equations which describe their behaviour as a function of various parameters. For example, the circuit described at the transistor, resistor and capacitor level could be simulated by the

continuous simulator. The behaviour of all these electronic components are governed by the mathematical formula.

Figure 2–5 shows an example of circuit while figure 2–6 displays the result of continuous circuit simulation. The simulation is done in 5Spice circuit analysis software[19].

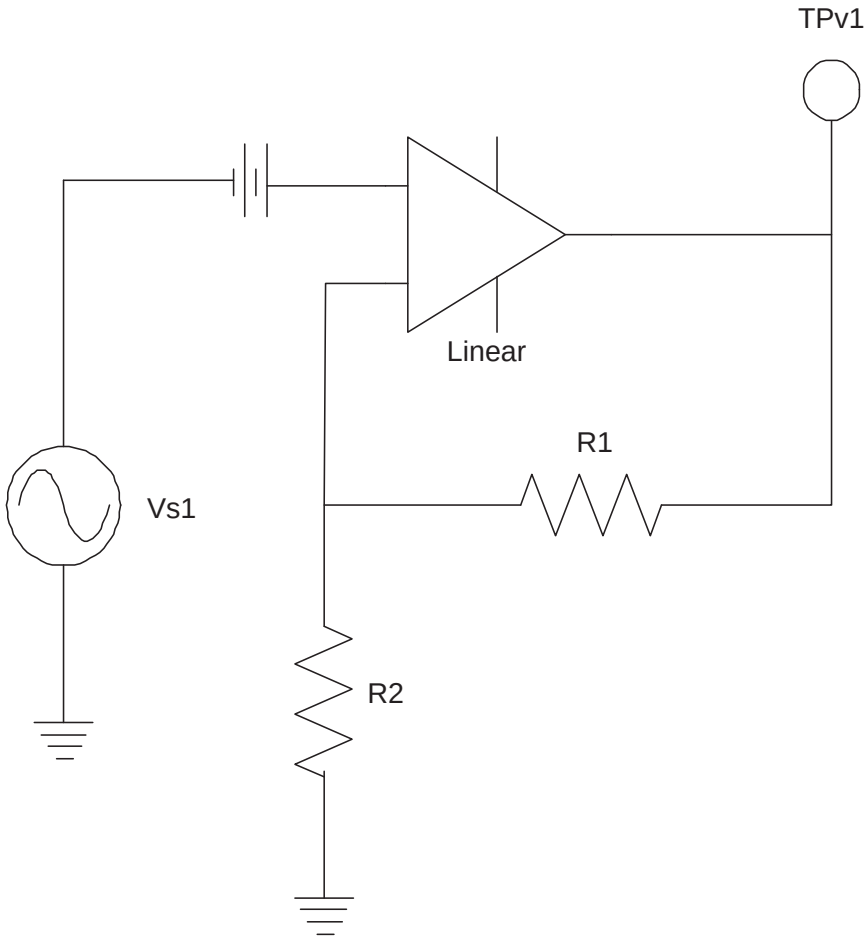


Figure 2–5: Analog circuit

Unfortunately the mathematical equations employed by a continuous simulation could be computationally intensive. Therefore, continuous simulation may be slow and

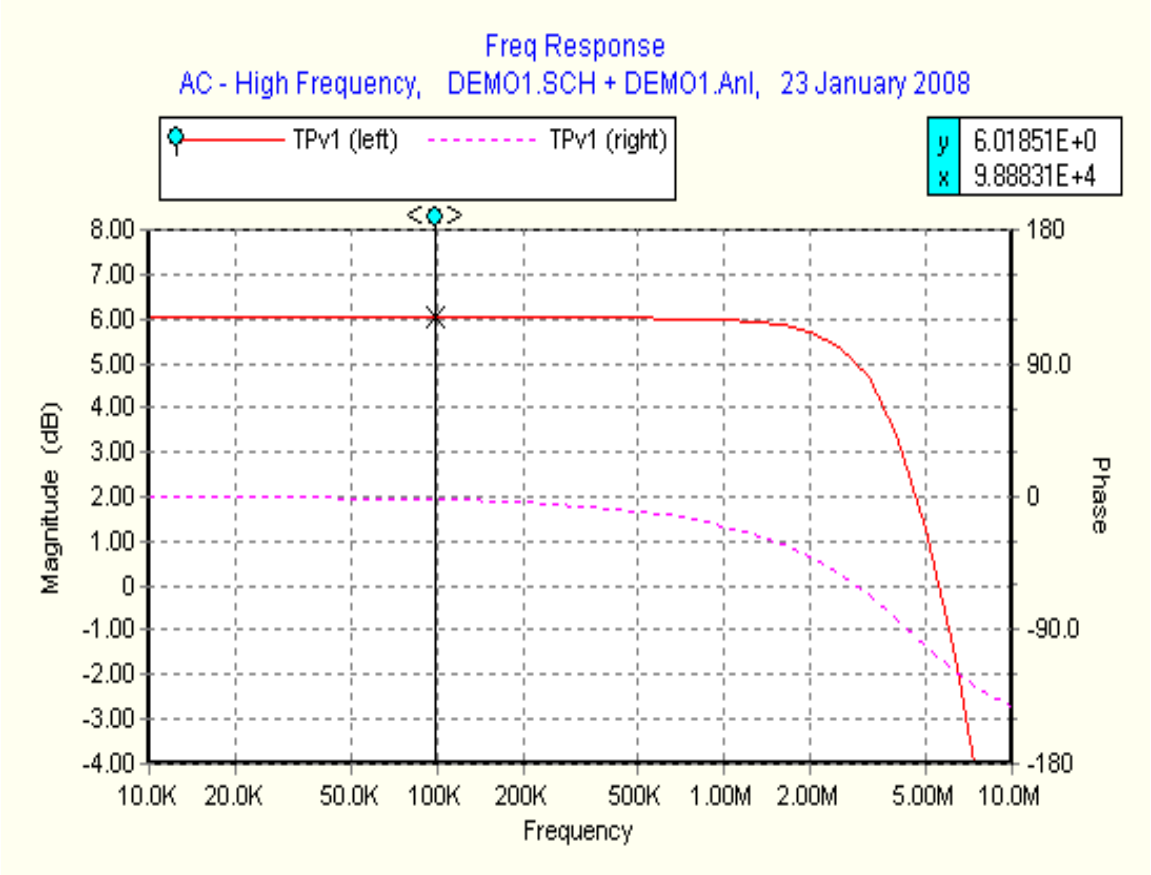


Figure 2–6: Analog simulation of the circuit

is only useful when simulating circuits which are described at a low abstract level such as analog level simulation[20].

In order to overcome the poor simulation speed of the continuous simulation, the discrete simulation is introduced, which is usually faster while providing a reasonably accurate approximation of a circuit system's behaviour.

Discrete simulation is divided into two subtypes, time-driven discrete simulation and event-driven discrete simulation.

Time-driven discrete simulation uses uniform time increments or ticks to advance simulation. Smaller time ticks could produce better precision while slowing down the simulation at the same time. At each simulation tick, the attributes of the models need to be evaluated. For example, in simulation of the trajectory of a projectile, the position and velocity are calculated in each tick by using the forces acting upon the projectile.

Discrete event simulation (DVES) [17] describes a simulation system in which only events can cause the simulation state to change. An event causes a change of the simulation state. In between events, the state of the simulation does not change. This allows a more efficient simulation than a continuous simulation (or time-driven discrete simulation) because the system state is only evaluated as the result of an event being executed.

Event scheduling normally uses two data structures. One is the state, while the other is the event queue, which is ordered by the timestamps of events. New events are inserted at the bottom of the event queue. An event scheduling algorithm operates by removing events from the event queue and processing them until the simulation finish time is reached or the event queue is empty. As a consequence of processing an event, new events can be scheduled and inserted into the event queue. The timestamp of the processed event is used to advance the simulation time. The simulation algorithm is shown in figure 2–7.

2.5 Logic Simulation

We start the introduction of Verilog simulation from logic simulation of digital circuits since Verilog simulation is actually one type of logic simulation.

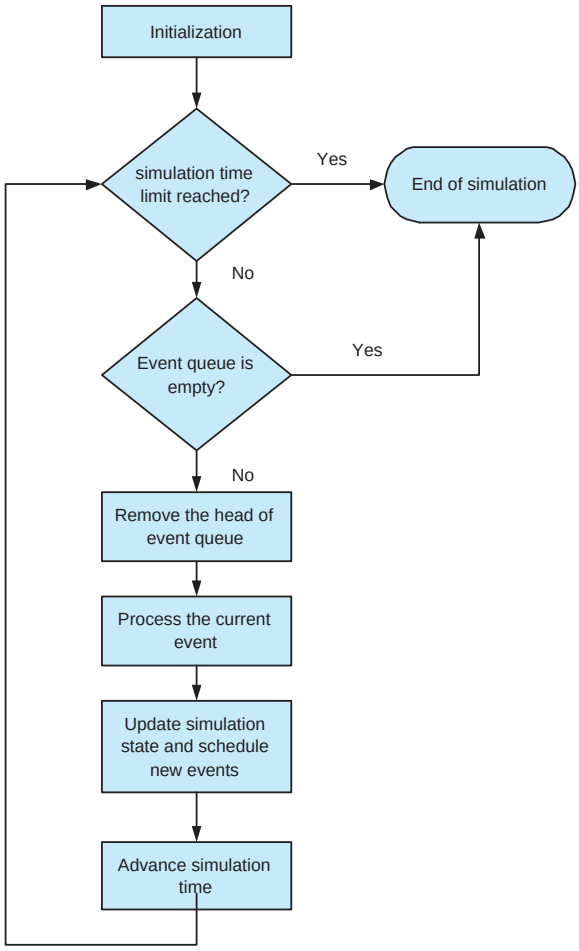


Figure 2–7: Discrete event simulation algorithm

Logic simulators are in widespread use as tools used to analyze the behavior of digital circuits. Logic simulators are used in hardware design verification to verify logical correctness and to perform simple timing analysis of logic circuits.

Logic simulators are also used for fault analysis[9]. The simulation could dump the waveform which logs the activity of the related signals in the digital circuits. With the waveform, the ASIC designers could locate the bug in the digital design from simulation without any electronic instrument such as oscilloscope and logic analyzer.

value	*Purpose*	*Value Encoding*
0	Forcing zero	00
1	Forcing one	01
X	Forcing unknown	10
Z	High impedance	11

Table 2–1: The logic state and its purpose

2.5.1 The Logic Simulation Model

The basic components of a logic circuit are predictably logic gates -AND, NAND, NOR and OR gates. The circuit is described by a graph or a hypergraph in which the nodes represent gates and the links represent wires. The nodes are modelled by software processes, referred to as logical processes (LPs) in the distributed simulation literature. Incoming channels of an LP correspond to the fanin list of a logic gate while the outgoing channels correspond to its fanout list.

The logic simulation model uses a finite set of values to represent the type of signal propagating throughout the circuit. The 4 values that a signal may have are presented in table 2–1.

A signal change is modelled as an update event containing a timestamp, source and destination gates, an identification and a value which corresponds to the new value of the wire. When an LP receives an update event, it sets its local clock to the the timestamp of the event, evaluates its output and schedules the resulting output change(s) as update events for its fanout list.

2.5.2 Discrete Event Logic Simulation

Figure 2–8 represents a simple logic circuit comprised of three gates. The circuit has three inputs (A, B and C), one output (F) and two internal wires (D and E).

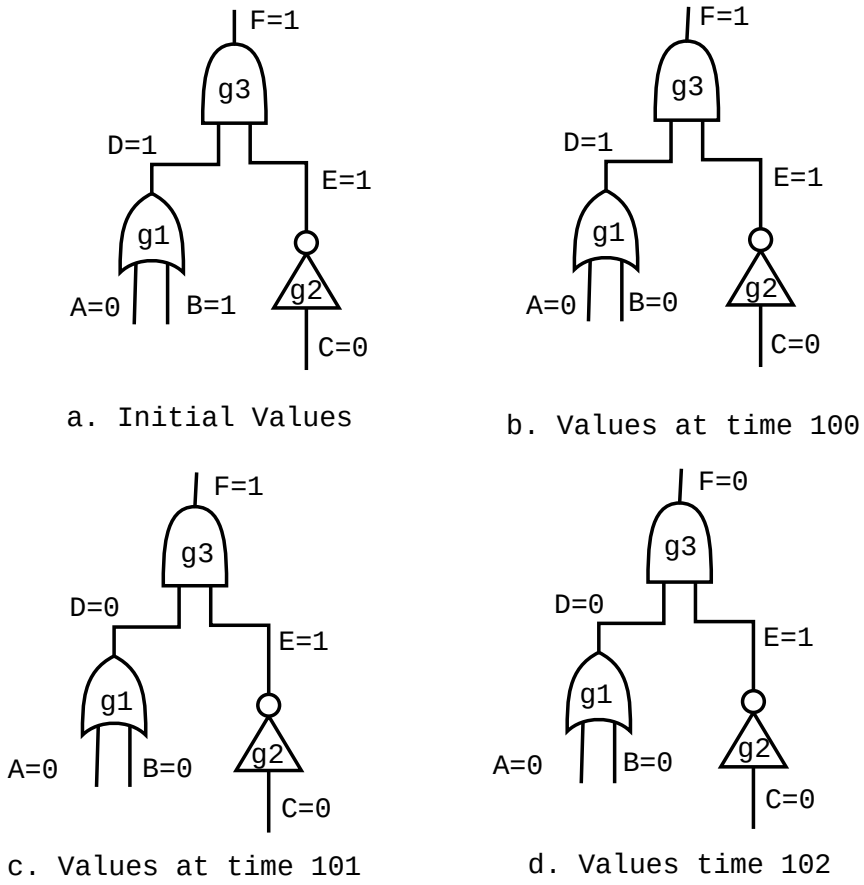


Figure 2–8: Logic simulation of a digital circuit

Assume that each gate has a unit delay, i.e. the simulation advance time at each gate is one time unit for each event. Initially, the gates have the values shown in Figure 2–8.a. An event occurs on wire B at time 100, changing it from 1 to 0 as shown in Figure 2–8.b. At time 100, gate g1 is evaluated to see if there's a change on its output D. Since D will change from 1 to 0, this event is scheduled in the future.

At time 101, gate g1's output D will be set to 0 as indicated in Figure 2–8.c and this new value will be propagated to the gates on g1's fanout, g3. Then g3 is evaluated

to see if there will be an output change on F. As can be seen in Figure 2–8.d, F will change from 1 to 0.

2.6 Event-driven Verilog simulation

In Verilog, before simulation begins, the design hierarchy is first elaborated . This means all the pieces of the Verilog code (modules/primitives/instances) are put together. The elaboration is similar to linking of the C language. The simulation cycle is then continuously repeated during which events are processed and signals are updated. A Verilog simulation cycle consists of the steps as shown in figure 2–9. The pseudo code of Verilog simulation comes from the Verilog language reference manual[21]. The interested readers should refer [21] for the details.

Time in Verilog simulator has two dimensions, the simulation and delta cycle. Delta cycle is used to distinguish those event with the same timestamp.

In Verilog it is possible to assign a delay mechanism to an assignment statement. Transport delay is characteristic of wires and transmission lines. Inertial delay models the real behavior of logic gates. The timestamp of new scheduled events for a node is the current simulation time plus the delay of the node.

```
while (there are events) {
 if (no active events) {
    if (there are inactive events) {
       activate all inactive events;
    }
    else if (there are nonblocking assign update events) {
       activate all nonblocking assign update events;
   } else if (there are monitor events) {
       activate all monitor events;
   } else {
       advance T to the next event time;
       activate all inactive events for time T;
   }
 }

 E = any active event;

 if (E is an update event) {
  update the modified object;
  add evaluation events for sensitive processes to event queue;
 }
 else { /* shall be an evaluation event */
  evaluate the process;
  add update events to the event queue;
 }
}
```

Figure 2–9: Verilog simulation pseudo code

CHAPTER 3

Parallel/Distributed logic simulation

3.1 PDES: Parallel/Distributed discrete event simulation

Parallel and distributed simulations are widely used to speedup large scale simulation applications. They differ in the computing platform used. A parallel simulation runs on multiprocessor machines in which communication is fast and the memory is usually shared between processors. On the contrary, a distributed simulation runs on separated computer systems connected with network in which the communication overhead is significantly larger than multiprocessor and each computer has its own memory. In the thesis, our computing platform is computer systems connected with Gigabit network so we call our simulation system as distributed simulation system.

In general, parallel/distributed simulation consist of logical processes(LPs) that represent physical processes of the modelled system. Each LP simulating a portion of the modelled system generates, sends and receives events to and from each other. Thus each LP handles both events generated locally and events triggered by other processes. An LP has an input queue in which event are stored in increasing timestamp order. As in sequential simulation, events are processed in strictly increasing timestamp order.

An LP stores its state and maintains the local virtual time (LVT) which is the current simulation time of the LP. The overview of a parallel/distributed system is depicted in figure 3–1.

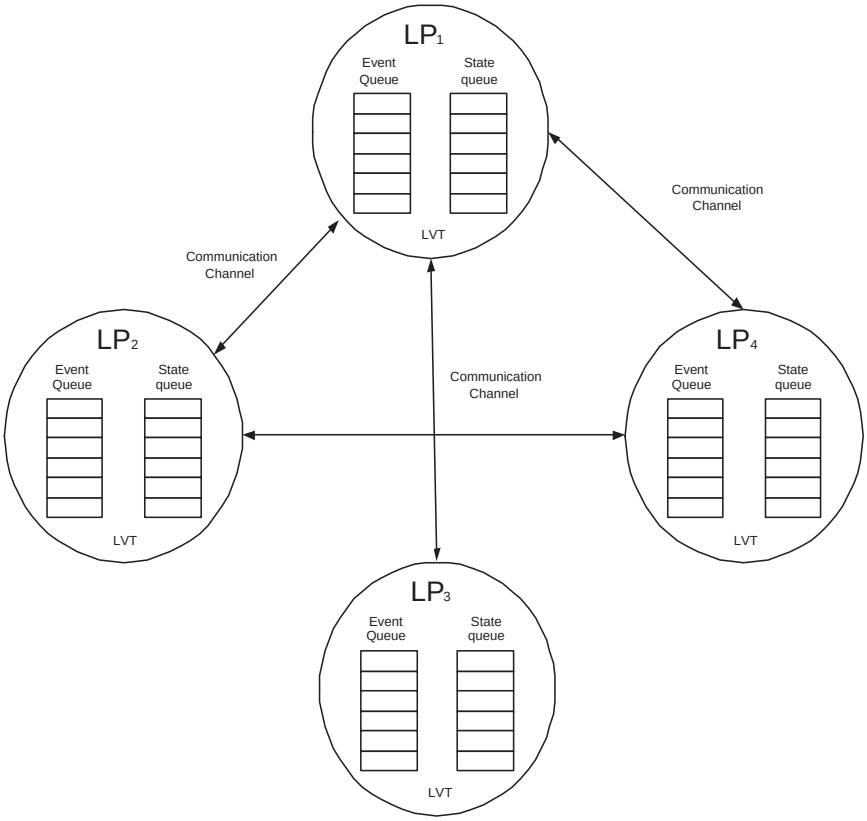


Figure 3–1: Overview of the parallel/distributed system

Causality is the central issue of the distributed simulation. In order to guarantee the correctness of the distributed simulation, it is necessary to preserve the event causalities across LPs. Lamport[22] suggested the notion of a logical clock which is a monotonically increasing counter in order to maintain causality. Each logical process maintains its own logical clock whose value is used to add a timestamp to the events

sent by the process. Lamport also introduced the happened-before relation. It is known as causal ordering which is based on the following two simple intuitive points.

- If two events occur in the same process, they should occur in the order in which the process observes them
- Whenever a message is sent between processes, the event of sending the message always occurs before the event of receiving the message

With the concept of the logical clock and the logical time, physical time can be abstracted since simulation could be guaranteed correct if causality order is maintained. According to how the causality constraint is dealt with, there are two major categories of parallel and distributed simulation protocols, the conservative and optimistic approaches. The conservative approaches process only those events that are guaranteed unable to affect other LPs while optimistic algorithms allow speculation and recover from any resulting causality violations.

3.2 Conservative Synchronization

The algorithms described in [23] were perhaps the first synchronization for parallel/distributed simulation. The conservative algorithms are distinguished by their blocking behavior when there are no safe events to process. Safe events are those events such that the simulator is guaranteed not to receive an event with a smaller timestamp. Conservative LPs can execute safe events in increasing timestamp order but must block when there is no safe events.

The advantage of this algorithm is that it is easy to implement and the overhead is very low. However, the blocking behavior somehow limits the concurrency of the parallel/distributed simulation. Another drawback of the algorithm is that deadlock

may occur if a collection of LPs are all waiting for a message. Thus, conservative algorithms require a method to either avoid or to detect and break deadlock.

The most common approach to avoid deadlock is the use the null message[23]. Each time an LP sends a message to another, it also sends a ”null message” to all other LPs with the same timestamp plus ”lookahead”.

Lookahead is critical to the performance of the distributed simulation with conservative synchronization. Fujimoto [24] defines lookahead as follows:

”Lookahead characterized the ability of a process to predict future messages that it will send based on knowledge of messages it has already received. In particular, if a process has received all messages with timestamp t or less, and can predict all future messages with timestamp t+lh or less, we say the lookahead of the process is lh”

If the lookahead is poor, the event population will be decreased in the simulation thus the parallelism is reduced since the event processing is delayed and few events are sent out.

The null message is used to notify the receiving LP that it will not receive any messages earlier than the null message from the source LP. Based on the knowledge of LVT plus lookahead from every neighboring LPs, the LP could determine which events are safe to execute.

Null message can create a huge communication overhead, especially in a distributed simulation environment. There have been many attempts to reduce the number of null messages. In [25], the approach is to only send null messages upon request. Whenever an LP is blocking, it sends a request message to its neighboring LPs and then waits for a responding null message which will unblock it. The authors of [26] proposed another approach in which the timestamp and lookahead values are stored

separately within null messages. The lookahead value is then sent to other LPs in the piggyback mode. Both normal messages and null message could carry the lookahead value and relay it to the other LPs.

Deadlock detection algorithm[27] deadlock breaking algorithm[28] makes use of knot detection. A knot is defined as a subgraph such that every node in the subgraph can be reached from every other node in the subgraph and no node outside the subgraph is reachable. A knot in the subgraph implies a deadlock in distributed simulation system.

Deadlock breaking algorithm[28] forces the event with the smallest timestamp in all LPs inside the deadlock knot.

In order to overcome the excessive amount of performance-degrading communication caused by deadlock prevention algorithms[27, 28], [29] proposed a protocol that attempts to balance the need for deadlock prevention synchronization information with the cost of providing the information. The author [29] claimed that the protocol is not only more efficient but also can ensure time accuracy.

3.3 Optimistic Synchronization algorithm: Time Warp

The most widely known optimistic algorithm is Time Warp. Similar to conservative algorithms, the parallel/distributed system utilizing Time Warp consists of LPs which communicate by messages. Each LP advances its simulation until it detects a violation in local causality. LPs detect violations in causality when they receive a message with a smaller timestamp than their LVT. The message with the smaller timestamp is referred to as a straggler. In Time Warp, LPs perform a rollback operation in which the LP reverts its state to the most recent LVT which had a correct causality and then

resumes the simulation from that point. In order to rollback to a previous state, the states of the LP need to be saved periodically (one possibility is after each event). This is referred to as checkpointing.

Time Warp also needs to save all of its output messages. When a rollback happens, the LP sends out ”anti-messages” corresponding to the output messages which were previously sent. The anti-messages are used to annihilate the output messages at their destination LPs. Due to the overhead of state saving and output event queue, Time Warp requires a good deal of memory.

The following diagram 3–2 illustrates the components of a simulation system with Time Warp.

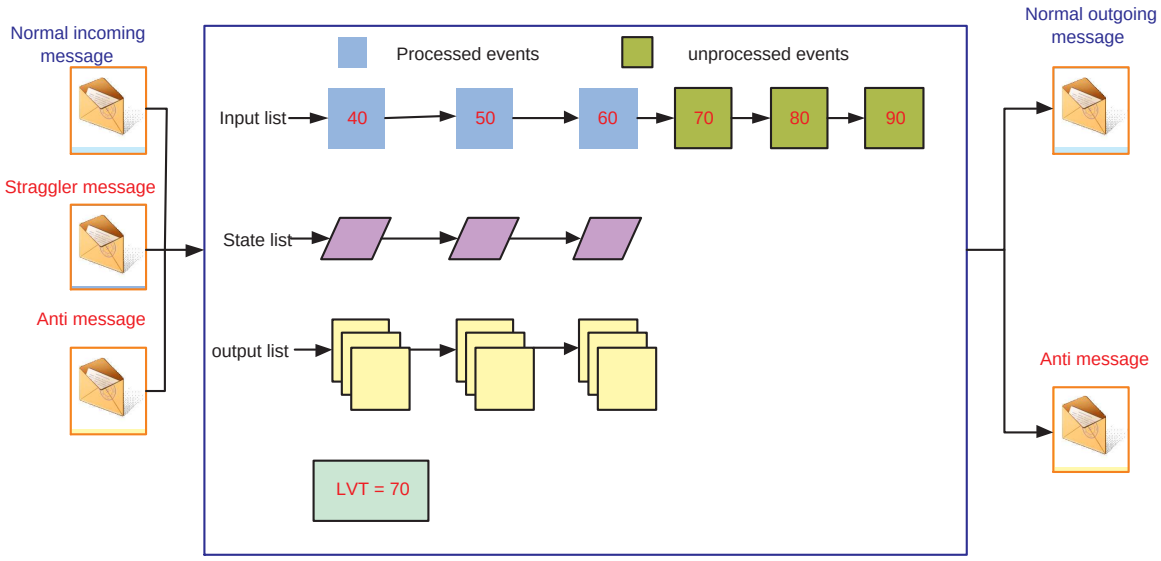


Figure 3–2: Components of a logical process with Time Warp

LPs detect violations in causality when receiving a message whose timestamp is smaller than their LVT. The message with the smaller timestamp is referred to as a

straggler. When an LP receives a straggler, the rollback process begins. The process could be summarized as the following steps.

1. State restoration: The LP restores its state to an element in the history event list which has a smaller timestamp than the straggler. The LP then frees the memory occupied by the states which have timestamp larger than the restored state.

2. Anti-messages: The LP sends out anti-messages for each of the elements in the output event queue which were sent after the timestamp of the restored state. The anti-messages are used to annihilate the corresponding message in the destination LP or behave as a straggler to cause the destination LP to rollback.

3. Resume simulation: The LP resumes simulation from the timestamp of the restored state.

The overhead of rollback is huge in terms of inter-processor communication traffic and the computation related to state restoration and anti-messages. Furthermore, the rollback of one LP may cause further rollbacks of other LPs in a chained reaction. This phenomenon is known as cascading rollback[30]. Another interesting problem is called ”dog chasing its tail”[31] which can be briefly explained as an erroneous computation wave circling among a few logical processes at a rapid rate. The rollback and cancellation wave is some distance behind the erroneous computation and is trying to outrun it. However, if the rollback and cancellation wave cannot spread faster than the erroneous computation wave, the erroneous computation may never be caught so the simulation will be stuck in the rollback wave

3.4 State of the art of Time Warp

Due to the memory overhead and rollback explosion associated with Time Warp, a lot of research[32, 33, 34, 35, 36] has been done to alleviate their effects.

3.4.1 Rollback reduction

The authors of [32] proposed lazy cancellation. In lazy cancellation, the propagation of anti-messages is delayed until the simulation is resumed after rollback and reaches the LVT of the earliest message in the output message list. At that point, if the newly generated messages differs from the messages in the output list, anti-messages are sent out. Since only the delta of the anti-messages are sent out, lazy cancellation avoids unnecessary anti-message traffic. The overhead of the lazy cancellation is the anti-message list.

3.4.2 GVT and fossil collection

Memory consumption for the history states could be huge. In order to free the memory occupied by the history states which have no use anymore, Global virtual time (GVT) was introduced by Jefferson[8] as follows.

”The GVT at real time r is the minimum of (1) all virtual times in all virtual clocks at time r, and (2) the virtual send times of all events that have been sent but have not yet been processed at time r.”

Since no LP would be able to rollback to a time prior to the GVT, each LP could release all states and events earlier than GVT. Events with timestamp less than GVT is referred to as committed events. The procedure to release memory occupied by the committed states and events is called fossil collection.

The GVT algorithm is relatively easier in a shared-memory environment than in a distributed-memory environment since there are no events in transit that have been sent out by the source process and have not been received by the destination process. In the shared-memory environment, the minimum of the local virtual time of all logical processes is GVT. However, in a distributed-memory environment, the events in transit makes GVT calculation more difficult. The naive way of calculating GVT in distributed-memory environment is to stop the simulation and restart it after the calculation is done. But this is too expensive so the preferred solution is to obtain an estimate of GVT. The estimation of GVT provides a lower bound on the smallest time stamp of all events no matter they are in transit or waiting to be processed.

[37, 38] solve the problem of events in transit by acknowledging each received event. However, this approach results in large message traffic and could degrade the simulation performance significantly.

Asynchronous token-passing algorithms[39, 40, 41] have been proposed to address the message traffic problem. A token is passed around the processes and the distributed GVT calculation is divided into two phases: the start phase and the stop phase. In the start phase, an initiating process P0 initiates the GVT computation and sends out START token. When a process receives the START token, it forwards it to its successor in the virtual ring topology and starts keeping track of the smallest timestamp of all messages it is sending. After the START token returns to the initiator, the stop phase is launched. The initiator sends a STOP token containing its smallest timestamp which is the minimum of the LVT and the timestamps of events in transit since the START phase. When a process receives a STOP token, it compares its smallest timestamp with the timestamp of the STOP token and sends out the STOP token with the smaller

timestamp to its successor. When the initiator receives back the STOP token, the timestamp associated with the STOP token is the new GVT, which will be broadcast to the rest of the processes.

The decentralized GVT algorithm is described by [36], which is based on the distributed snapshots which utilizes the following elements:

- Cut point is an instant separating computation into past and future
- Cut is a set that consists of a cut point for each LP
- Cut message is a message that crosses a cut from past to future
- Cut value is the minimum among the timestamps of both cut messages and all cut points along a cut

Figure 3–3 depicts the relationship between processes and events with a cut for GVT computation. Each horizontal line shows the time of the processes. The circle represents an event and the arrows show the path of causality between events. A cutline divides the events into two disjoint sets, the events occurring before the cut line is defined to be the events in the past while the events occurring after the cut line events in the future.

The implementation of a cut in Mattern's algorithm[36] uses the colouring scheme. Initially all processors are coloured white. A white process sends only white events and a red process sends only red events. Every process counts the number of white events it sends and receives. A red process keeps track of the smallest timestamp of all red events it sends. The algorithm is described as follows,

- 1. The initiator start GVT computation by sending a cut event to its successor and change its color to red

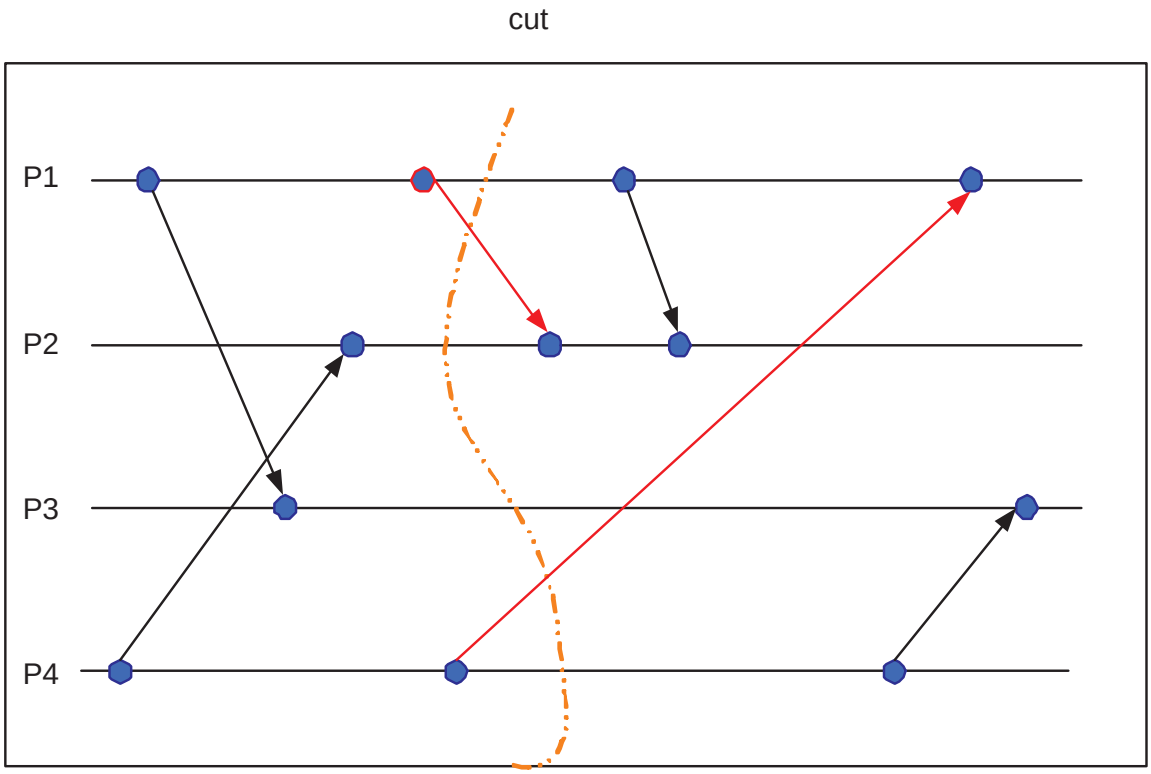


Figure 3–3: A time diagram with a cut

- 2. The receiving process of the cut event will forward it to its successor in the ring and changes its color to red
- 3. When the initiator receives back the cut event, the timestamp of the cut event is the minimum of the LVT of all processes and the smallest timestamp of red events sent by each process

In order to avoid the possibility of an event causing a rollback after it reports its minimum time, at least two cuts are required. The second cut has to be placed ”far enough” to the right of the first cut. Mattern[36] accomplishes this through the use of a vector counter. Every process Pi maintains a vector counter to keeps track of the number of white events it has sent to Pj. Every time a white event is received, the

process decrements its vector count V[i]. The cut event will accumulate the vector counters of each process as it goes around the ring of the processes. At the end of the first round, the accumulated vector counter indicates the number of events in transit. If some white events are still in transit, which means the accumulated vector counter is not zero, the second cut is initiated. In this round, the cut event waits at each process until all white events are due at that process. When the cut events returns again to the initiator, the GVT is calculated to be the minimum of the smallest LVT of all processes and the smallest red event timestamp as carried by the cut event.

3.4.3 Other memory saving techniques

A lot of approaches have been developed in order to reduce memory consumption even more, besides the fossil collection. The details of periodic check pointing [42], incremental state saving[42], reverse computation[43] and rollback relaxation[44] will be described in chapter 5. In chapter 5, we also describe a new approach, event reconstruction to reduce memory consumption.

There are several approaches[8, 32] designed to recover the parallel/distributed simulation system when the system runs out of memory. All of them are based on the rollback model and are invoked when an optimistic simulation is either out of memory or cannot use fossil collection to reclaim memory anymore. Message Sendback[8] and Gafni's protocol[32] return messages whenever the distributed simulation system runs out of memory. Message sendback[8] returns the unprocessed input messages back to the sender thus releasing the memory occupied by the messages. [32] expands message sendback by taking into consideration recovering memory from the input message list, output message list and history state list. The unprocessed input message

will be returned to the sender. The output messages are also removed by sending the corresponding anti-messages while the history state will also be purged.

Cancelback[45] and artificial rollback[45] are similar to Gafni's protocol[32] in that both of them reclaims memory from input/output event list and the history state list. Instead of sending back input messages, it forces the system to a previous state which has available memory, even if there is no causality violation. Artificial rollback[45] identifies the LPs that are further ahead in simulation time and rolls them back. Cancelback[45] rolls back the simulation indirectly by sending messages to senders indicating that they must reverse certain messages.

3.5 Parallel and distributed logic simulators

In his PhD thesis[46], Briner developed a parallel logic simulator based on Time Warp. He makes use of incremental state saving[34], a bounded time window[35] and different synchronization granularities in his simulator. Briner also points out the necessity of efficient partitioning algorithms. Briner achieved the speedup of 23 over sequential simulation on 32 processors of a BBN GP1000 system, running mixed level simulations.

Matsumoto and Taki describe a parallel gate-level simulator[47] based on Time Warp, which obtained more speed-up compared to an asynchronous conservative algorithm and to a synchronous method. As a result, they argue that Time Warp is superior to these methods. An improvement to Time Warp is to send only one antimessage to the affected LPs during rollbacks. This message is the one with the smallest timestamp of all of the antimessages that are sent when using aggressive cancellation.

Manjikian and Loucks[48] implemented a parallel gate-level simulator on a network of workstations. They used a hybrid approach for synchronization of the LPs. Individual LPs run in an optimistic way but event messages are only sent to other LPs when they are safe. According to the authors, an important role is played by partitioning algorithms. They used cone partitioning[49] with enhancements in order to incorporate estimated circuit behavior in the partitioning algorithm. Speedups between 2 and 4.2 are achieved on 7 processors from ISCAS89 benchmark circuits. The highest speedup of 4.2 was achieved through cone partitioning algorithm.

Bauer and Sporrer realized a parallel logic simulator[50] based on Time Warp. They used the sequential event-driven gate-level simulation LDSIM[51] as a base for their work. The authors propose incremental state saving to keep the memory overhead low. Luksch implemented a parallel version of LDSIM on the Intel iPSC/860 hypercube[52, 53]. The authors indicate that there may be a huge amount of state information that has to be stored during the course of a simulation. LDSIM achieved speedups between 2 and 4 over the sequential simulation on 12 processors on circuits with 3,500 to 19,200 gates on ISCAS89 benchmark circuits.

Bagrodia [54] developed a parallel gate-level circuit simulator in the Maisie simulation language [55] and implemented it on both distributed and shared memory parallel architectures. They achieved speedup of about 3 on 8 processors of a Sparc1000 for the conservative protocol and about 2 for the optimistic protocol on the four largest ISCAS85 benchmark circuits with gate numbers of 1193, 1667, 2307 and 2418. The K-FM[3] partitioning algorithms were used to partition the circuits.

L. Zhu implemented a parallel logic simulator for million-gate VLSI circuits[56]. The authors claimed that they achieved superlinear speedup for up to 17 processors.

The circuit used is synthesized netlist of the Viterbi decoder and the partitioning algorithm is hMetis[6] developed at the University of Minnesota.

Avril's CTW(Clustered Time Warp)[57] is a hybrid algorithm which makes use of Time Warp between clusters of LPs and a sequential algorithm within the cluster. The authors claimed that CTW[57] is useful in logic simulation of digital circuit where there are a number of LPs having low computational granularity. In the thesis, we extended CTW into an object-oriented version[1] and integrated it with Icarus[58], an open source Verilog simulator.

Kim [59] developed a parallel logic simulator on MIMD distributed memory machines. A new partitioning algorithm, improved Concurrency Preserving Partitioning (iCPP) [60] was proposed. Event-lookahead Time Warp (ETW) which is the hybrid integration of event-lookahead conservative protocol and the Time Warp protocol was proposed and implemented on an IBM SP2 parallel machine with 10 processors.

Discovery[61] developed at the Ohio State University is a framework for parallel and distributed simulation of digital and analog VLSI systems, in which the digital portion is described in VHDL, while the analog portion is described in SPICE[62]. The simulator made use of optimistic synchronization for digital components and conservative synchronization for analog components. A naive partitioning algorithm is used which allocates an equal number of LPs to each processor.

In xtw[63], a new event scheduling mechanism XEQ and a new rollback procedure rb-messages are proposed for use in optimistic logic simulation. XTW groups LPs into clusters, and makes use of a multi-level queue, XEQ, to schedule events in the cluster. Experimental results over large circuits (5-million-gate to 25-million-gate) shows XTW scales well with both the size of circuits and the number of processors.

3.5.1 Parallel and distributed Verilog/VHDL simulators

SAVANT[64] was developed at the University of Cincinnati. It consists of three major components, SAVANT, TyVIS and Warped. Warped is an optimistic parallel discrete event simulator based on Time Warp. TyVIS is a VHDL kernel which provides the runtime support for the simulation of VHDL designs. It operates on top of Warped. SAVANT compiles VHDL source code and generates C++ code, which is compiled and linked with TyVIS and the Warped library to obtain the final simulation executable.

The TyVIS library is also referred to as the TyVIS kernel. It is an extension of the Warped kernel inheriting all of its attributes and methods. The generated code is comprised of class instantiations and function calls provided by the kernel.

Experiments for several partitioning algorithms, including a multi-level algorithm[65] were reported, with the multi-level algorithm resulting in the fastest simulation times.

Meister [66] developed a framework called DVSIM for a parallel event-driven simulator of VLSI designs described in VHDL. Both conservative and optimistic synchronization protocols were implemented. The simulator evolved from the sequential simulator VSIM developed by Levitan[67]. The experiments were done on ISCAS89 benchmark circuits with gates 892, 15709 and 40685. The authors pointed out that there was no speedup for the small circuit. But for larger benchmark circuits, the speedup was about 4 on 12 processors.

Tun [68] presents a parallel Verilog simulator - PVSim, which is based on optimistic asynchronous parallel simulation algorithm and MPI library. A new module-based simulation component mapping method is proposed. And an efficient module-based partition algorithm combined with pre-simulation partition algorithm is adopted.

CHAPTER 4

DVS: An object-oriented framework for distributed Verilog simulation

This chapter describes the architecture and implementation of the distributed Verilog simulator, DVS. The research described in this chapter first appeared in [1].

4.1 Overview of Icarus Verilog

Icarus Verilog [58] is an open-source EDA (Electronic Design Automation) Verilog simulator being developed by Stephen Williams. As shown in figure 4–1, Icarus Verilog includes two independent parts: an IVerilog compiler and a VVP(Verilog Virtual Processor) simulator. The bridge connecting these two parts is VVP assembly code, an intermediate representation of the original circuit. The IVerilog compiler is a translator that translates the input Verilog source code into VVP assembly code. The VVP simulator is an event-driven simulation engine, which interprets VVP assembly code and process the events. We give a brief introduction to Icarus Verilog in the following sections.

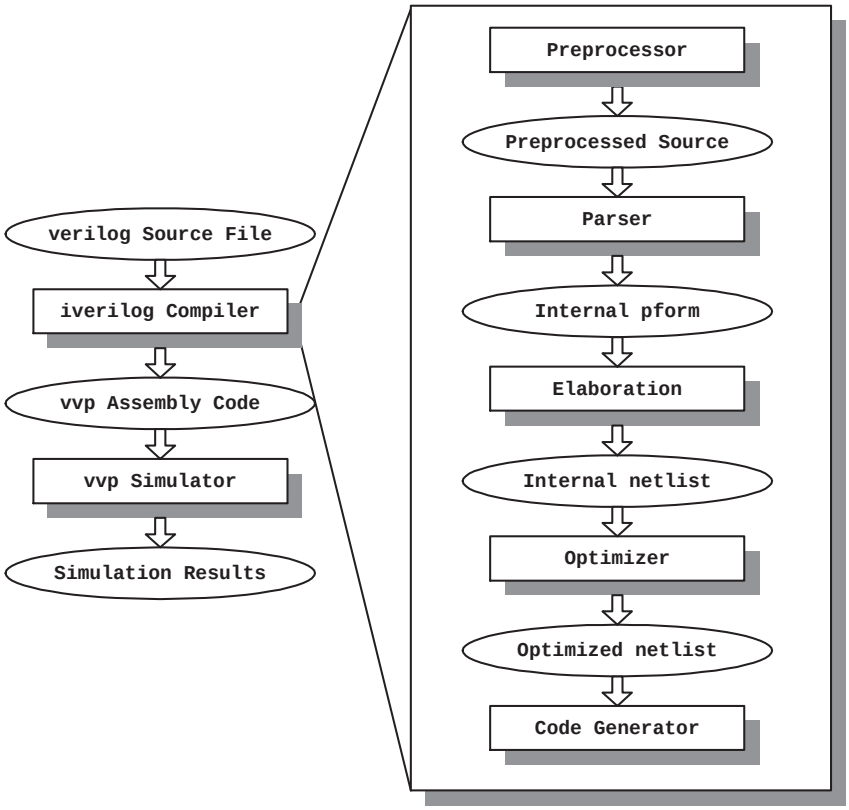


Figure 4–1: Architecture for Icarus Verilog

4.1.1 IVerilog Compiler

Although the Verilog language enhances modularity and encapsulation by the use of modules in the source file of a circuit, the hierarchical structure of modules is not appropriate for the purpose of simulation. The IVerilog compiler flattens modules in the original source file in the following five consecutive phases:

- Preprocessor

  It mainly performs file inclusion for ’include directive and macro substitution for ’define directive. For each include directive, the preprocessor reads the include file and places it into the original source file at the location of the

include directive. The preprocessor also substitutes text macros defined by define directives. Finally, in order to display error messages, the preprocessor generates line directive to print the line number of the source file.

- Parser

  The preprocessed source file is parsed and its internal representation is generated with syntax and semantic checking performed.

- Elaboration The root module is located, unresolved references are resolved, and all instantiations of modules are expanded. After scope elaboration and netlist elaboration, an internal flattened netlist is generated from the hierarchically structured modules.

- Optimizer

  Some useful transformations can be performed on the internal netlist in order to simplify netlist and improve simulation efficiency.

- Code generator

  All circuit information is now stored in the flattened and optimized internal netlist. There are five target formats that can be generated from the netlist, of which VVP assembly code is the default one used for simulation.

4.1.2 VVP Simulator

The VVP simulator is an interpreter for VVP assembly code. It parses VVP assembly code to generate netlist of structural items and exert input vectors to drive the simulation.

The separation of the IVerilog compiler and the VVP simulator is similar to the separation of compiler and interpreter in Java. The VVP assembly code is the

counterpart of bytecode in Java. Since large VLSI circuit files normally take a long time for compilation, this strategy saves a lot of time. Once the VVP assembly code file is generated by the IVerilog compiler, we can use it in our partitioner and distributed simulator.

In the following sections, we explain our effort to design and implement DVS, an object-oriented framework for distributed Verilog simulation.

4.2 Architecture of DVS

Figure 4–2 illustrates the architecture of DVS. It takes VVP assembly code as input, which is generated by the IVerilog compiler for simulation efficiency. The VVP parser constructs the functor list and virtual thread list, which will be used by the distributed simulation engine after partitioning.

The 3 layers of DVS are shown in the right side of figure 4–2. The bottom layer is the communication layer which provides a common message parsing interface to the upper layer. Inside this layer, the software communication platform can be PVM or MPI. Users can choose one of them without touching the code of upper layer.

The middle layer is a distributed discrete event simulator, OOCTW, which is an object-oriented version of Avril's CTW(Clustered Time Warp)[57]. It provides the following services to the top layer.

- sending and receiving positive messages or anti-messages
- rollback LPs after receiving a straggler or an anti-message
- state saving and restoring
- GVT computation and fossil collection

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

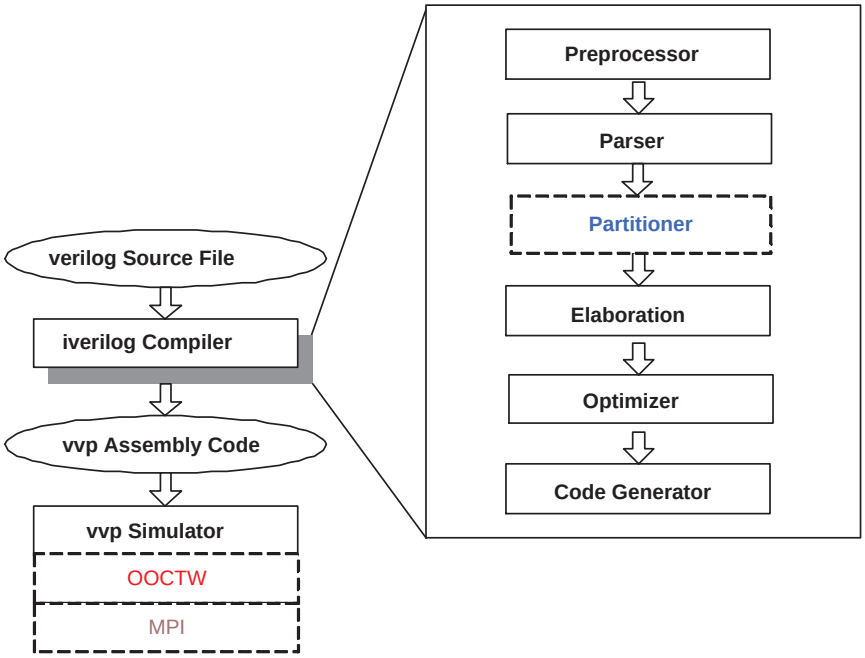


Figure 4–2: Architecture of DVS

4.3 VVP parser

The Verilog language provides the ability to model a circuit by means of both structural descriptions and behavioral descriptions. Structural descriptions model the circuit as a network of interconnecting gates and wires, while behavioral descriptions model the circuit at a higher level as *always* and *initial* blocks. They are translated to *.functor* statement and *.thread* statement in the VVP assembly code generated by the IVerilog compiler. The VVP parser parses VVP assembly code and instantiates these

structural and behavioral statements as functors and vthreads which are described in the following sections.

4.3.1 Structural item: functor

Structural items are represented by functors in the VVP simulator. Each functor has four input ports and one output port. Gates with more than four input ports are divided into smaller gates and cascaded. Functors also have associated delay values. All functors are stored in a functor list which will be used for partitioning and simulation.

During the simulation, when the value in any input port of a functor changes, a new output value is calculated by querying a truth table. If the result is different from the current value in the output port, the value in the output port is updated, and a propagation event is scheduled with the associated delay value. After this delay time expires, the propagation event is processed, and the signal is assigned to corresponding input ports of all fanout functors.

4.3.2 Behavioral item: vthread

Behavioral items are represented by virtual threads (vthread) in the VVP simulator. It should be noted that vthreads run in the virtual machine of the VVP simulator instead of running directly in the operating system. Each vthread contains a mechanism for thread execution, including a program counter, 4 numeric index registers and 64k private bit registers.

All vthreads instantiated by the VVP parser are organized as a vthread list. In gate-level logic simulation, vthreads are normally used to drive functors with input vectors.

4.4 Partitioner

Partitioning plays an important role in affecting the performance of the distributed logic simulation[69]. In order to exploit different partitioning algorithms in DVS, we designed a generic partitioner and integrated it into the framework of DVS.

4.4.1 Design of Partitioner

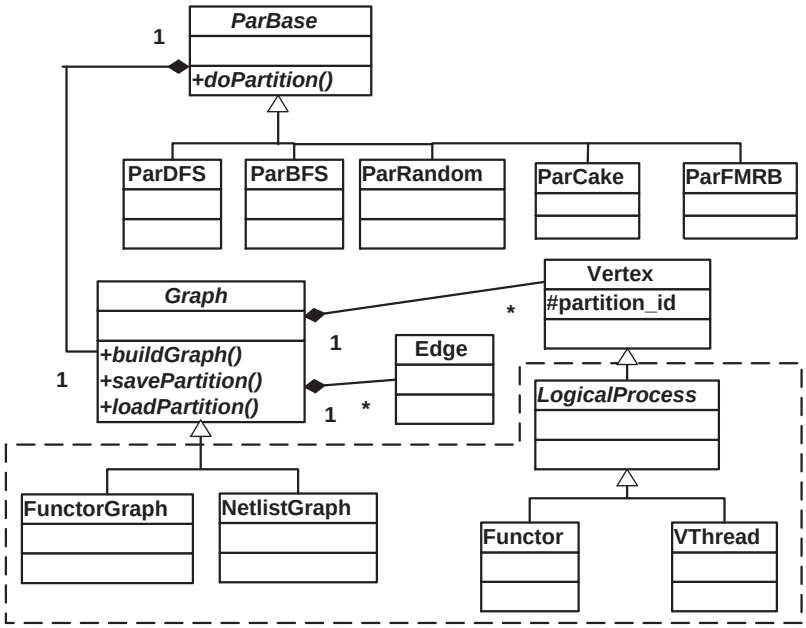


Figure 4–3: UML description of partitioner

The design goal of our partitioner is to provide a flexible infrastructure for testing different partitioning algorithms applied to different circuit implementations. As shown in figure 4–3, the partitioner has two major parts: the partitioning algorithm and the circuit graph being partitioned.

The circuit graph is represented by Vertex and Edge objects in the abstract Graph class. The Graph class also provides interfaces to partitioning algorithms for retrieving

information for vertices and edges in a graph. Designers of different simulators can subclass it and implement the buildGraph method to fill in vertices and edges using application-specific information. In DVS, we use FunctorGraph to build the graph using the functor list.

The base class for partitioning algorithms, ParBase, is also an abstract class. All partitioning algorithms should be derived from ParBase and provides an algorithm-specific implementation for the doPartition method. In DVS, the partitioner will automatically select the corresponding algorithm at run time based on the partitioning argument in the command line.

4.4.2 Partitioning functors and vthreads

Since circuit information is available in both the IVerilog compiler and the VVP simulator, we can perform partitioning on either side. After investigating the internal data structures on both sides, and also considering that both functors and vthreads are LPs in DVS, we decide to use the functor list and vthread list in our partitioning algorithm.

The structure of the functor list is similar to an adjacency list, which is convenient for partitioning. Furthermore, since every computer in the simulation has the same copy of functor list, it can be readily used for message routing when the destination functor resides on remote computer. If dynamic load balancing is performed during the simulation, the re-partitioning can be done on the functor list, and the re-mapping of functors is as simple as modifying the partition-id of corresponding functors.

The treatment of vthreads is different from functors. We observe that when functors and vthreads are placed in the same partition, more rollbacks tend to occur.

OOCTW uses clustered rollback, i.e., a straggler at one LP causes all LPs in the same cluster to rollback. Vthreads tends to advance much faster than functors in LVT because behavioral simulation is more efficient than logic simulation. Thus a fast vthread is likely to cause all of the slow functors in the same cluster to rollback more frequently. Therefore, we put all of the vthreads on one computer. Since the total number of vthreads is small in gate-level logic simulation, the lost concurrency can be compensated for by fewer rollbacks. The large number of functors are partitioned and assigned to the rest of the computers in the simulation.

4.5 OOCTW(Object-oriented CTW)

4.5.1 Motivation

Clustered Time Warp(CTW)[57] was developed with logic simulation in mind. LPs (representing gates) are grouped into clusters. Each cluster has an input and an output queue associated with it. Events were executed sequentially within the cluster. Several rollback and checkpoint algorithms were developed for use with CTW.

CTW is a good starting point for the implementation of object-oriented Time Warp. A cluster bundles gates together in order to overcome the fine event granularity of VLSI simulation. Furthermore, a cluster provides a very good basis for load balancing. We can also move an entire cluster between processes instead of just moving gates. However, CTW is not object-oriented. It is not easy to integrate it directly with the sequential simulator. Therefore, we used an object-oriented paradigm to transform CTW into OOCTW, which (we hope) will be an open and flexible synchronization backend.

The main design goal of OOCTW is to integrate it with the original Verilog simulator. The motivation for the design is to limit the changes made to the sequential simulator because we hope to take advantage of its new version. The other design goal is to make the Time Warp library more reusable, readable and understandable so new members in the laboratory can concentrate on the optimization algorithms instead of falling into the black hole of Time Warp. Finally, the Time Warp library must be flexible and open so it can be a test bed for new optimization algorithms.

To date we have only implemented one of the rollback algorithms developed for CTW, clustered rollback, in OOCTW. In clustered rollback, when a straggler or an antimessage arrives at the cluster, all of the LPs with larger LVTs than the straggler or the antimessage are rolled back. Other modifications of CTW are checkpointing when the LVT of an LP advances and the use of Mattern's GVT algorithm[36].

4.5.2 Class hierarchy of OOCTW

The diagram above the dashed rectangle in figure 4–4 is a UML description of OOCTW. The Cluster is the container and scheduler of all LPs. The scheduling algorithm we employed is LTSF(Lowest Timestamp First). An LP is scheduled for execution when it has an event with the lowest timestamp in the cluster. The cluster manages a future event list and an output event list. The GVT computation is also processed in the cluster. Each time the cluster receives a new GVT, it invokes fossil collection. Statistics are also collected in the Cluster such as simulation time, rollback number, communication cost, etc.

As shown in figure 4–4, class LP executes rollback and provides virtual methods for state saving and state restoration. The derived classes override the virtual methods

to have application-specific implementations of state saving and restoration. An LP maintains a processed event list but doesn't maintain an output event list. When an LP sends out an event which crosses the cluster boundaries, it inserts a copy of the event into the output event list of the cluster.

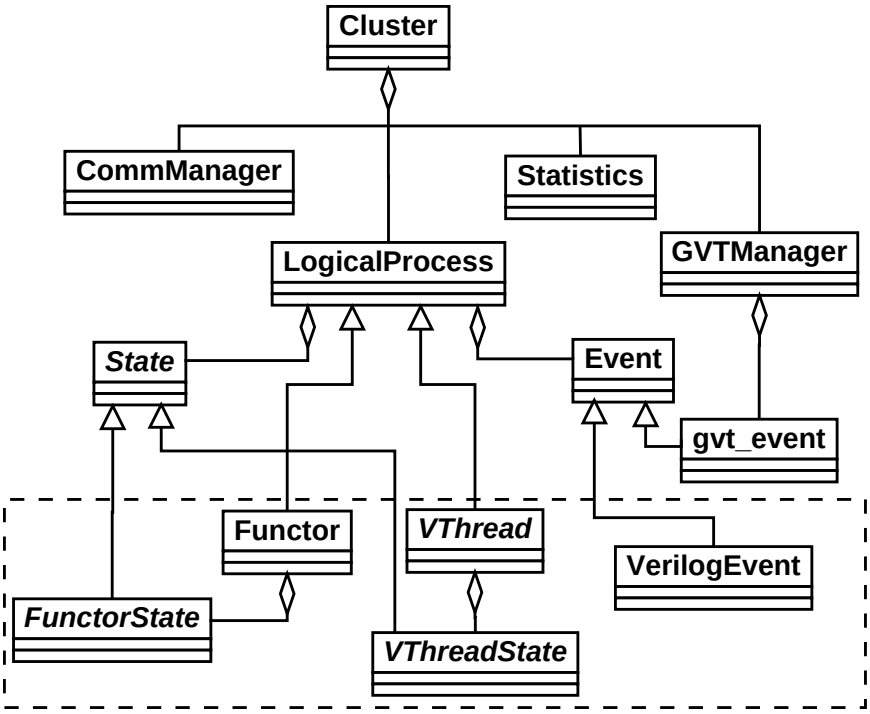


Figure 4–4: UML description of OOCTW

The members of the event class include the sender and receiver of the events, the sending and receiving time, the sign of the event and the ID of the events. The event class provides operators such as $\ll$, $\gg$ and == to compare the timestamp of two events. The procedures to decide whether an event is a negative event are also provided in the class. Class gvt_event inherits from event class. It is used to compute the GVT via Mattern's algorithm[36].

Type	*Usage*
THREAD	Schedule a virtual thread
EVAL	Evaluate the functor
PROP	Propagate the value change after gate delay
INQUIRY	Inquiry value of a remote functor
RESPOND	Respond inquiry of functor value
FINISH	Finish of the simulation

Table 4–1: Events in distributed Verilog simulation engine

The base class for state is an abstract base class. It provides an interface for the application specific state. In DVS, there are two different kinds of LPs with their own state, which will be explained in detail in the following section.

4.6 Distributed Simulation Engine

The original sequential VVP simulator is turned into a distributed simulation engine via its integration with OOCTW. The classes in the distributed simulation engine are shown in the dashed rectangle of figure 4–4. *Functor* defines structural items in Verilog while *Vthread* defines behavioral blocks. They both inherit from class LP and override the abstract member methods so they are able to save state, rollback and restore state. *FunctorState* and *VthreadState* implement the interface of *state*, which is used to log the state of the functors and vthreads.

VerilogEvent inherits from class *event*. Several types of events in the distributed simulation engine are shown in table 4–1.

THREAD event is used to awake the blocked virtual thread which is waiting for an event to happen, such as a value change of a register. *EVAL* and *PROP* are used to propagate value changes among the network of functors.

INQUIRY event is used to detect the value of a functor located in a remote host. For example, the variable ’a’ in statement *$display($time,,a)* may be located in a remote processor. Therefore, the virtual thread will send an *INQUIRY* message to get the value of the remote functor. The remote processor will send back the response as soon as it processes the event.

After partitioning, the simulator schedules the *THREAD* event to invoke the virtual threads whose partition ID matches the host id of the local machine. These virtual threads will feeds input vectors to the network of functors. The simulator keeps processing events until it gets FINISH event broadcasted by machine 0.

Each simulator in different machines keeps the topology of all of the functors in order to route messages. However, only those functors with the same ID as the local host are active. The passive functors are only used to route messages. No evaluation happens on passive functors.

The $display and $monitor in Verilog are used to print values of variables or logic gates. However, the state of an LP is not stable until its LVT is smaller than GVT. Therefore, I/O can’t be committed immediately after the instruction is issued. Hence, we created a delayed I/O instruction list to save all I/O instructions and the time at which they are issued. Each time a new GVT is generated, the simulator will check the delayed I/O list. If the timestamp of the I/O instruction is smaller than GVT, it will be committed.

4.7 Optimization to distributed Verilog simulation engine

- Direct execution of zero delay event

When the simulator generates a zero delay event which has the same timestamp as the current LVT, it executes the event directly without first inserting it into the event queue then popping it out and executing it. This introduces some in-determinism but doesn't affect the final simulation result. The direct execution reduces memory operations and speeds up the simulation. In fact, there are a lot of simultaneous events in the Verilog simulation. A functor will propagate its value change to all of its fanout functors. All propagated evaluation events are simultaneous events which have the same timestamp as the current LVT because we assume zero wire delay. If the fanout functors resides on the same cluster, the Verilog simulator can execute the corresponding evaluation events directly.

- on-the-fly fossil collection

  In order to improve the efficiency of the simulator, Icarus simulator maintains a free event list in order to minimize the invocation of the system calls such as malloc/free and new/delete. Each time the simulator schedules a new event, it first checks the free list. If it is not empty, the new event can directly use the memory space occupied by the head of the free list. When the simulator finishes processing the event, it puts the event pointer into the free list instead of deleting the memory space.

  The free list is inherited in the distributed simulator. Moreover, we created the free state list for state saving of LPs.

4.8 Preliminary Experiments

All of our experiments were conducted on a network of 8 computers, each of which has dual PentiumIII processors and 256M RAM. They are interconnected by a

Myrinet(www.myri.com), a high speed network with link capacity of 1Gbit per second. All machines run the FreeBSD operating system. LAM MPI is used for message passing between different processors.

The Verilog source file used in the simulation describes a 16bit multiplier. It includes 2416 gates and one virtual thread which feed 50 random vectors to the circuit. We assume the unit gate delay and zero transmission delay on the wire. Only the simulation results with BFS partitioning is presented because we have compared the performance of BFS, DFS and random partitioning algorithm and found BFS to have the best performance. BFS partitioning algorithm can reduce communication, which is the most expensive operation in distributed environment. Each data point collected in the experiments is an average of five consecutive simulation runs. The number of machines in the figure doesn't include machine 0 which only contains vthreads. The simulation time for 1 machine is the running time of the DVS without partitioning.

The simulation time vs. the number of machines is shown in figure 4–5. It should be noticed that the simulation time is longer when 2 machines are used. This is caused by the load imbalance and communication cost. From the upper part in figure 4–6, we know that the partitioning algorithm only reduces the total number of event processed on machine 1 by a small amount when 2 machines are used. However, the communication cost increases by a large amount. The total communication cost can be computed by multiplying the number messages shown in the lower part of figure 4–6 with average sending/receiving cost, which is listed in table 4–2. The reduction in workload is not large enough to compensate the communication cost. Therefore, the total simulation time for 2 machines is longer than the time for 1 machine.

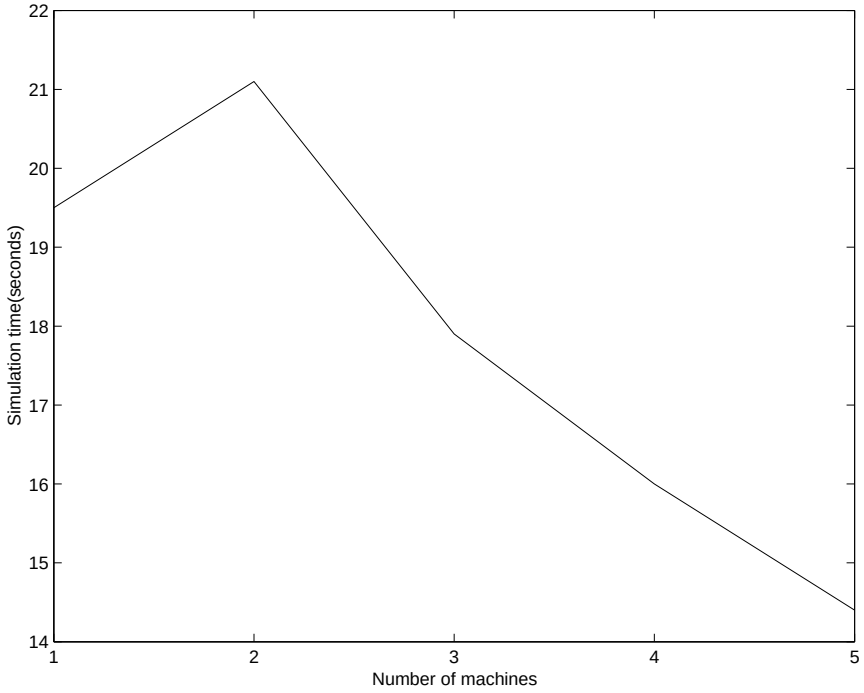


Figure 4–5: Simulation time in seconds vs. number of machines

Operation	*Time*
Processing an event	1.83us
Saving a state	2.08us
Saving an event	2.56us
Sending a message(Blocking)	31.9us
Receiving a message(Blocking)	32.2us
Message latency	10us

Table 4–2: Cost of operations in DVS

Using more machines reduces the number of events processed per machine a great deal, thus the time used to process events is reduced by the amount which is large enough to compensate the communication cost involved in the distributed simulation. The simulation times keep decreasing when the number of machines increases from 3 to 5. We get a speedup of 1.4 when 5 machines are used.

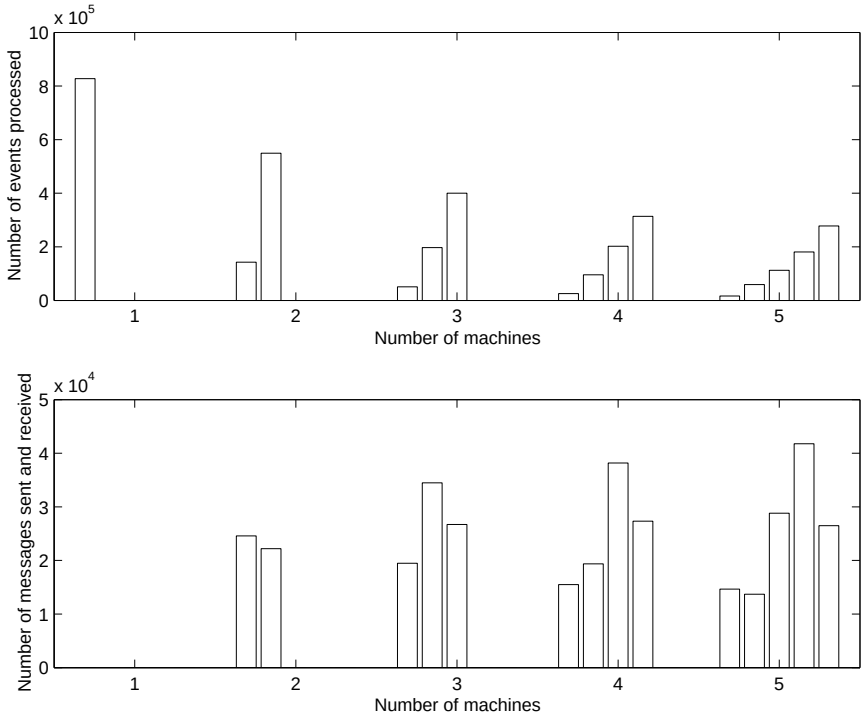


Figure 4–6: Number of events processed by every machine(Upper part) and number of messages sent and received(Lower part) by every machine vs. number of machines. Note: The two figures use different scale.

Unfortunately, so far DVS still runs slower than the original Icarus Verilog simulator. We attribute this to the fine granularity of VLSI simulation, large communication cost, load imbalance and the small circuit size of our Verilog source file. From table 4–2, we know that overhead for VLSI simulation is more than 2 times the cost of processing an event.

By increasing the event granularity, reducing communication costs and achieving load balance, we look forward to outperforming the original simulator in further experiments(in which we simulate larger circuits) and demonstrating the scalability of DVS as well. The following chapters will detail our effort to do so.

CHAPTER 5

Event Reconstruction in Time Warp

The research described in this chapter first appeared in [70].

5.1 Introduction

Time Warp is known for its relaxed synchronization so that out-of-order processing is possible with the help of a rollback mechanism. In the rollback, the LP restores the previous state and sends out anti-messages to cancel events generated as a consequence of the causality errors. The advantages of Time Warp are that the LPs never have to block in order to guarantee only safe events could be processed so that causality errors cannot occur. However, the disadvantages are additional costs associate with the rollback mechanism.

State saving mechanism is an essential part of a Time Warp system. It is necessary for the distributed simulation system to save enough state information in order to guarantee that any state that is possible to restore in case of rollback could be reconstructed. The naive implementation of state saving could be to save state for each event processed. This approach is usually referred to as copy state saving(CSS). However, this could be too costly in terms of both memory consumption and simulation performance.

In a successful distributed simulation environment, the number of events rolled back are usually much less than the total number of events processed. Hence, it is often wasteful to save a complete copy of each state for each event since most states will never be used for rollback purposes. Based on this assumption, several methods have been proposed to reduce the state saving overhead. These methods could be roughly classified into two categories: sparse state saving(SSS) and incremental state saving(ISS) detailed as below.

- Sparse State Saving(SSS) [71, 72]

  Sparse state saving is also referred to as infrequent or periodic state saving. The state is not saved each time an event is processed. In case of rollback, the state is restored by retrieving the last state checkpointed before the rollback point. Then all the intermediate events between restored state and rollback point will be executed in order to restore the state at the rollback point. The reexecution of events is also referred to coast forward. During the coast forward phase, no anti-messages will be sent out since coast forward only serves to restore the state. The checkpointing interval could be static or dynamic. Static checkpointing interval is to save state every n^{th} events processed while dynamic checkpointing interval is calculated on the fly during the simulation.

- Incremental State Saving(ISS)[33]

  In many distributed simulation environment, such as large communication system or battlefield simulations, the state size could be huge while only a small portion of the state is updated after one event is processed. In such distributed simulation systems, it is often to use incremental state saving since it could be too expensive or even impossible to save the complete state of the system.

The idea of incremental state saving is to only log the changes to the state in the backtrail. Prior to the state change, its old value and address are logged. During state restoration in the rollback, the backtrail is traversed in the reverse order, from the most recent event to the rolled-back event, by writing the old values back into the associated addresses.

The incremental state saving schemes depict a trade-off between execution efficiency and programming transparency. A major drawback to incremental state saving(ISS) is the need for programmer awareness. State logging calls need to be explicitly inserted in the distributed simulation model.

5.2 Related work

A number of algorithms have been proposed to reduce the memory overhead caused by state saving, including incremental saving[33], checkpointing[71, 72], reverse computation[43] and rollback relaxation[44].

P.A. Wilsey[42] presents a comparative analysis of four approaches to dynamically adjusting the checkpoint interval and proposes an algorithm for dynamic checkpointing. The algorithm tries to balance the time spent saving state versus the time spent coasting forward. The goal of the algorithm is to minimize the time for state saving and coasting forward and to adjust the checkpoint interval accordingly. In our experimental section5.5, we compare the performance of event reconstruction and this heuristic algorithm.

Checkpointing results in a lower memory consumption and an improved execution time. However, it is difficult to achieve the optimal frequency of checkpointing. Dynamic checkpointing can be used to alleviate this problem. However, it faces the

problems of choosing tuning parameters, including the initial checkpointing frequency, the average cost of event processing and the average cost of coasting forward.

Reverse computation[43] computes state variables by reversing the operation sequence applied on the variables. It uses compiler-based techniques to generate the reverse computation code automatically. As a result, its implementation is more complex, although it is able to provide a significant performance improvement over checkpointing.

In rollback relaxation[44] all LPs are classified into two categories, memoryless and LPs and LPs with memory. A memoryless LPs' output is determined by the values of its inputs. Therefore, no state is saved for memoryless LPs. Instead, the LP reconstructs any required input state from the events of the input queue. The rollback relaxation mechanism is able to reduce the state saving overhead by a considerable amount in logic simulations because most LPs(AND, OR, XOR gates) in such simulation are memoryless.

5.3 Logic simulation and its characteristics

5.3.1 Characteristics of logic simulation

In this section we discuss the characteristics of logic simulation which inspired our work on event reconstruction. The detail explanation of the discrete event logic simulation could be found in chapter 2.

- Relatively small state size

  In the implementation of a logic simulator, such as DVS [1], the 4 signal values are encoded with two bits as shown in table 2–1. Every gate has up to four inputs and one output. Therefore, the state of a gate in Figure 5–1 includes ival

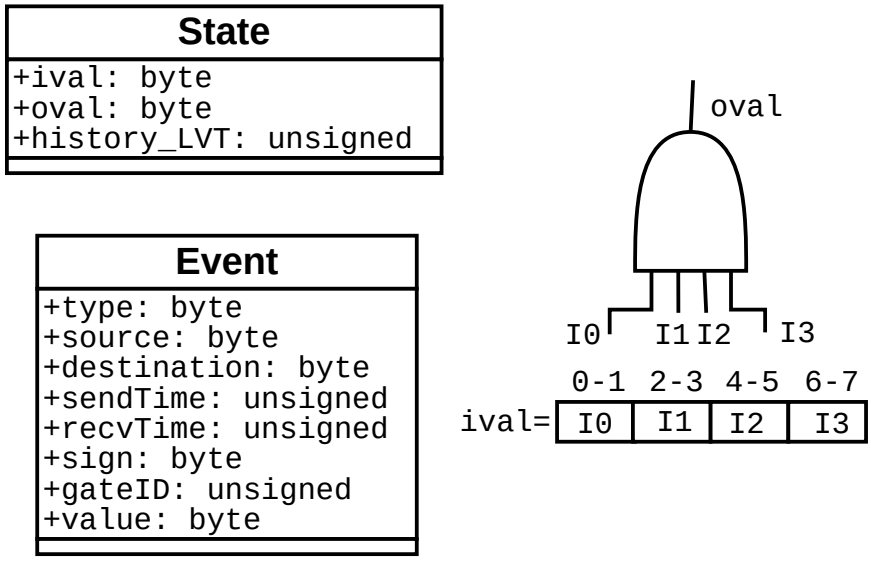


Figure 5–1: The size of the state and the event

and oval, each of which are one byte in length. The bits 0-1 are used to store the value of I0, bits 2 and 3 for I1, 4 and 5 for I2 and bits 6 and 7 for I3, as shown in Figure 5–1. For example, if ival is equal to 00001001, we know that I0 is equal to logical value 0, I1 logical value 0, I2 logical value 'x' and I3 logical value 1. This compact storage helps to save memory. The size of the state is only 16 bytes in DVS[1]. However, checkpointing has its greatest value when the size of the state is large.

- Large event size

  Figure 5–1 shows the structure of an event. The size of an event is 56 bytes, almost four times of the size of a state. Therefore, event saving causes at least 3.5(56/16) times more memory to be used than state saving if state saving is done for every event processed. In order to underscore this point, the amount of

memory consumed in the simulation of a 16 bit multiplier is presented in our experimental section.

- Large event population

  The event population is large because of the large number of gates, each of which is mapped to an LP. For example, the event population is 8,129,815 for s38584(about 20K gates) when the clock is 500kHz and the number of random input vectors is 100. If every event has to be saved, the associated memory consumption will be very large.

- Fine event granularity

  Logic simulation is known for its fine event granularity. Thus, the performance of distributed logic simulation is especially sensitive to the overhead caused by state saving and event saving. Reducing this overhead would certainly be useful for performance improvement.

With these characteristics of logic simulation in mind, we decided to reduce the memory occupied by events instead of reducing the memory consumed by states. The following section describes our approach to event reconstruction.

5.4 Implementation of Event Reconstruction

In this section, we explain the implementation of event reconstruction in detail. The data structures and algorithms which comprise this approach are described below.

5.4.1 Data structure

The data structure for event reconstruction is shown in figure 5–2. In DVS, the Cluster is the container and scheduler of all of the LPs. An LP maintains a state list.

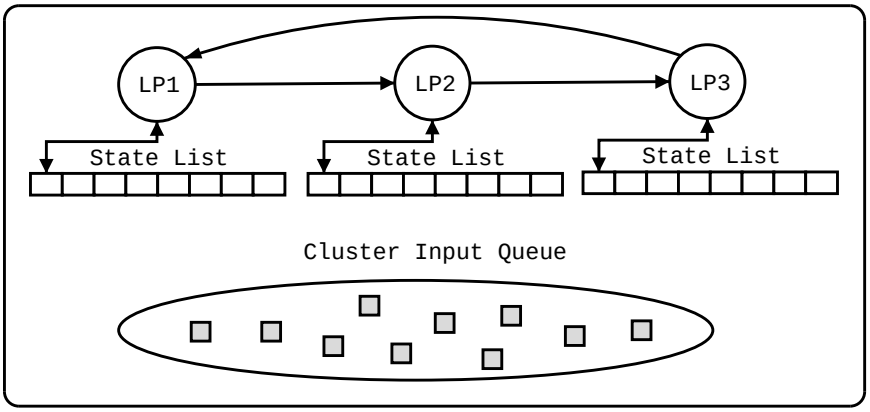


Figure 5–2: Cluster structure

In event reconstruction it is necessary to store all of the states at an LP. Since we build the input events and anti-events from the state list in our approach, *w*e don't need the input event list and output event list for every LP. The unprocessed events for all of the LPs in the same cluster are stored in a single priority queue data structure. The LP scheduling strategy is smallest timestamp first. We note in passing that the GVT computation also benefits from the single queue data structure.

5.4.2 Event annihilation

Time Warp uses a tuple (LPID, timestamp, eventID) to match positive events and their corresponding anti-events. The LPID is globally unique, indicating which LP will receive the event. The eventID is unique in the cluster, and is increased by one automatically whenever a new event is generated. The EventID is used, along with the timestamp to distinguish between simultaneous events. Unfortunately, the eventID is lost because we don't save input events in our approach. Instead, we use the signal value on the wire to compensate for the lost eventID information. The new tuple for event annihilation is (LPID, timestamp, signalValue). If both the timestamp and the

signalValue are the same for two events, they are considered to be identical events. If there exists more than one identical event in the event queue, the anti-event will pick the first one in the queue to annihilate. This approach introduces some indeterminism. However, Verilog[9] is a concurrent language, in which there are sources of non-deterministic behavior such as arbitrary execution order in zero time and arbitrary interleaving of behavior statements. Therefore, the simulation results are not guaranteed to be deterministic. In fact, simultaneous events are executed in arbitrary order in the Verilog simulator.

5.4.3 Port flag

We use a different rollback strategy for LPs inside the cluster and for LPs outside of the cluster. Inside the cluster, we roll back those LPs which are descendants of the LP which receives the straggler or anti-message. Anti-messages are sent to LPs outside of the cluster.

In order to implement the two rollback algorithms, a port flag is used for each LP port in order to indicate whether it is an internal port or an external port. The port flag is set at run time. Initially, every port flag is set to be an internal flag. When the LP receives an external message, it sets the corresponding flag to external. The implementation of the rollback algorithms making use of these flags will be explained in the following section.

5.4.4 Event builder

Only those events which change the input signals at a gate need to be reconstructed, as it is only these events which cause a change in the state of a gate.

Let s' be the state before the execution of event e and let s be the state after the event is executed. If s is equal to s', event e is considered null and need not be reconstructed. However, if s is different from s', event e can be rebuilt according to Formula 5.1. The signal value is the value on the wire, as shown in Table 2–1.

$$\begin{aligned} e.timestamp &= s.timestamp \\ e.signalValue &= s.signalValue \end{aligned} \tag{5.1}$$

For example, state s' is shown in Figure 5–3.a and state s in Figure 5–3.b. It is worthwhile noting that signal values on all ports are packed into a one byte state. By comparing the value on port I1 of states s and s', the event which happened at time 200 is reconstructed with the value on port I1 of state s, which changed from '00' in state s' to '01' in state s.

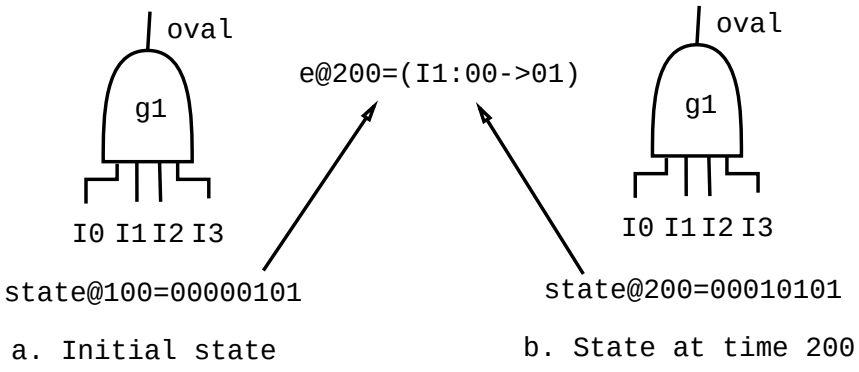


Figure 5–3: Event reconstruction

Input event builder

In Time Warp an LP saves events after processing them because if the LP rolls back, previously processed events will have to be reprocessed. Through event reconstruction, these previously processed events will no longer have to be saved. Instead, they are reconstructed by the input event builder, depicted in Figure 5–4.

```
input_event_builder(event* rb_event)
{
  reverse_iterator iter=state_list.rbegin();
  //main loop for the event reconstruction
  while((*iter)->LVT >= rb_event->recv_time)
  {
    state* s1 = (*iter);
    state* s2 = (*iter++);
    //only reconstruct the external event
    //ignore the reconstruction of internal events
    //Event reconstruction on port 0
    if (s1->ival&3 != s2->ival&3)
      if (s1->LVT == rb_event->recv_time || external_port_flag[0])
      {
        //reconstruct the event
        e->recv_time = s1->LVT;
        e->ival = s1->ival&3;
        if e->is_anti_event(rb_event)
          annihilate(e, rb_event);
        else
          schedule(e);
      }
    //Event reconstruction on port 1
    if ((s1->ival>>2)&3 != (s2->ival>>2)&3)
      if (s1->LVT == rb_event->recv_time || external_port_flag[1])
      {
        //reconstruct the event
        e->recv_time = s1->LVT;
        e->ival = (s1->ival>>2)&3;
        if e->is_anti_event(rb_event)
          annihilate(e, rb_event);
        else
          schedule(e);
      }

    //Event construction on port 2 & port 3
    //compare((s1->ival>>4)&3, (s2->ival>>4)&3)
    //compare((s1->ival>>6)&3, (s2->ival>>6)&3)
  }
}
```

Figure 5–4: Input event reconstruction algorithm

The algorithm loops through the state queue until the LVT of the state is less than the receive time of the event which causes the rollback. It picks a state s1 and its predecessor s2 from the state queue. If the input values of state s1 and s2 are different,

an event e is reconstructed according to Formula 5.1. The input values are bound into one byte. Therefore, the comparison is executed four times, once for each input port of the LP, as shown in the Figure 5–4. Moreover, due to the different rollback strategies for the internal events and external messages, we set a port flag to indicate the source of the events. For the external port, we reconstruct every event. However, we only reconstruct the events which have LVT equal to the LVT of the straggler event for the internal port. The reason for this is that the internal events which have a larger LVT than the straggler will be regenerated because of the cluster rollback strategy[57] used in DVS[1], which will rollback all LPs in the cluster. Therefore, the internal events will not be reconstructed because they will be regenerated by their source LP in the same cluster. The port flag is used to avoid unnecessary reconstruction of internal events. In fact, the algorithm of event reconstruction does not depend on the cluster rollback strategy. We are continuing to improve the rollback strategy and the event reconstruction algorithm. Further effort will focus on tree rollback instead of the cluster rollback. The tree rollback strategy only rolls back those LPs which reside in a tree whose root is the LP which receives the straggler event.

Anti-event builder

The anti-event builder works in the same way as the input event builder, as shown in figure 5–5. The anti-event is reconstructed by comparing the output values of two adjacent states, s1 and s2. After reconstruction, the anti-event is sent to those LPs which are in the fanout list of the current LP but not in the same cluster. For a cluster rollback, we don’t have to use anti-events to cause a rollback in the same cluster.

```
anti_event_builder(event* rb_event)
{
  reverse_iterator iter=state_list.rbegin();

  while((*iter)->LVT >= rb_event->recv_time)
  {
    state* s1 = (*iter);
    state* s2 = (*iter++);

    if (s1->oval&3 != s2->oval&3)
      {
        //reconstruct the anti event
        e->recv_time = s1->LVT;
        e->ival = s1->oval&3;
        e->flag = ANTI;

        for each external LP in fanout list
          send e to LP
      }
   }
}
```

Figure 5–5: Anti event reconstruction algorithm

5.4.5 Event processing loop

The basic algorithm for an optimistic LP is sketched in Figure 5–6. The LP removes the head event from the event queue and checks whether it is a normal event or a straggler or an anti- event. If it is a normal event, the LP first logs its state and processes the event. State is saved after every event. However, processed events will not be saved.

When the LP receives an anti-event or a straggler, it rolls back as in ”normal” Time Warp. However, the LP reconstructs the input events and output events from the state queue. This introduces a processing overhead which is similar to the cost for coasting forward in dynamic checkpointing.

```
while(GVT < FINISH_TIME)
{
  receive external events;
  pop an event from event queue;
  update LVT;
  if (event is straggler or antimessage)
    {
      input_event_builder();
      anti_event_builder();
      send_anti_events();
    }
  else
    {
      log_state();
      event_processing();
    }
}
```

Figure 5–6: Optimistic LP simulation algorithm

5.5 Experiments

All of our experiments were conducted on a network of 8 computers, each of which has dual PentiumIII processors and 256M RAM. They are interconnected by a Myrinet, a high speed network with link capacity of 1Gbit per second. All machines run the FreeBSD operating system while MPICH-GM is used for message passing between different processors.

The Verilog source file used in the simulation describes an ISCAS'89 benchmark circuit, S38584. It includes 19253 gates, 1426 D-type flip-flops and one virtual thread which feed 20 random vectors to the circuit. The clock frequency of S38584 is 1MHz. The other Verilog source file describes a 16bit multiplier. It includes 2416 gates and one virtual thread which feeds 200 random vectors into the circuit.

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

In the experiments, we compare the performance of DVS with dynamic checkpointing and with event reconstruction to that of "pure" Time Warp. The partitioning algorithm which we use is CAKE[73]. The dynamic checkpointing algorithm is initiated every 1000 events. Our event reconstruction algorithm requires that the state is saved after each event is processed.

5.5.1 Memory Usage

Memory consumption breakdown

The memory consumed by Time Warp is composed of the memory consumed by state saving and by event saving. Figure 5–7 presents the memory breakdown for the machine which has the maximum memory consumption. The data is collected for a 16 bit multiplier and for S38584 using Time Warp. The top of Figure 5–7 is the memory breakdown for the 16 bit multiplier with 200 random vectors while the bottom is the memory breakdown for S38584 with 30 random vectors.

We see from both of these that event saving consumes more memory than state saving. We define the *m*emory usage ratio to be the ratio of the memory consumed by event saving to the memory consumed by state saving and list these ratios in table 5–1 for the 16 bit multiplier and for S38584. We see that event saving consumes 4.73 times

circuit	*2*	*3*	*4*	*5*	*6*
16 bits multiplier	4.49	4.50	4.78	4.68	4.73
S38584	3.60	3.68	3.54	3.53	3.55

Table 5–1: The memory usage ratio

the memory used by state saving when 6 machines are used. On the average, *e*vent saving consumes almost four times the memory consumed by state saving.

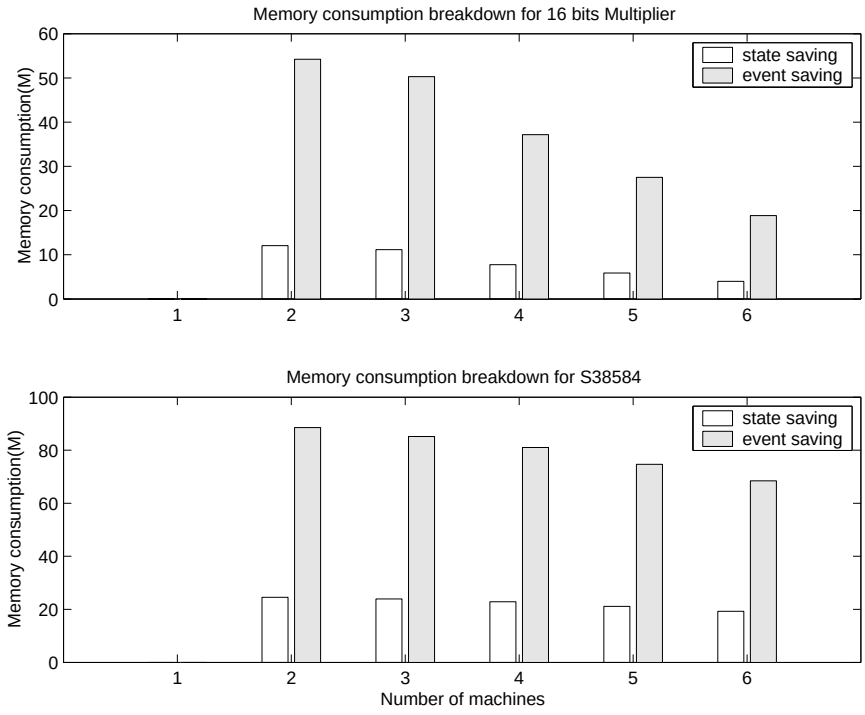


Figure 5–7: Memory consumption breakdown

Peak memory consumption

We define the *p*eak memory usage to be the maximum of all of the machines' maximal memory usages. Figure 5–8 shows the peak memory vs. the number of machines for the 16 bit multiplier. The memory used by one machine is only 0.45M because memory overhead is unnecessary. When two machines are used, event

reconstruction uses 1.79 times less memory than dynamic checkpointing and 2.34 times less than pure Time Warp. The ratio between event reconstruction and dynamic checkpointing decreases when more machines are used. The reason for this decrease is that the average number of events processed decreases when more machines are used, and consequently the memory occupied by event saving decreases. When 6 machines are used, event reconstruction uses 1.29 times less memory than is used by dynamic checkpointing.

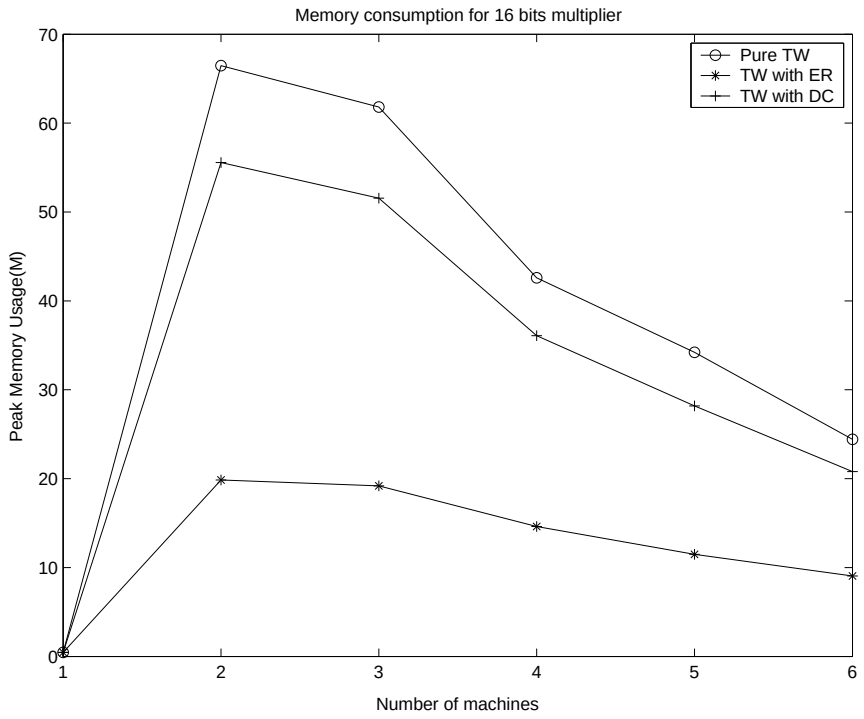


Figure 5–8: Memory consumption for 16 bits multiplier

Figure 5–9 presents the peak memory vs. the number of machines for S38584. Time Warp uses 119.06M when 2 machines are used. This leads to memory swapping and bad performance, as shown in Figure 5–11. Event reconstruction uses 54.21M when two machines are used, versus 99.18M by dynamic checkpointing.

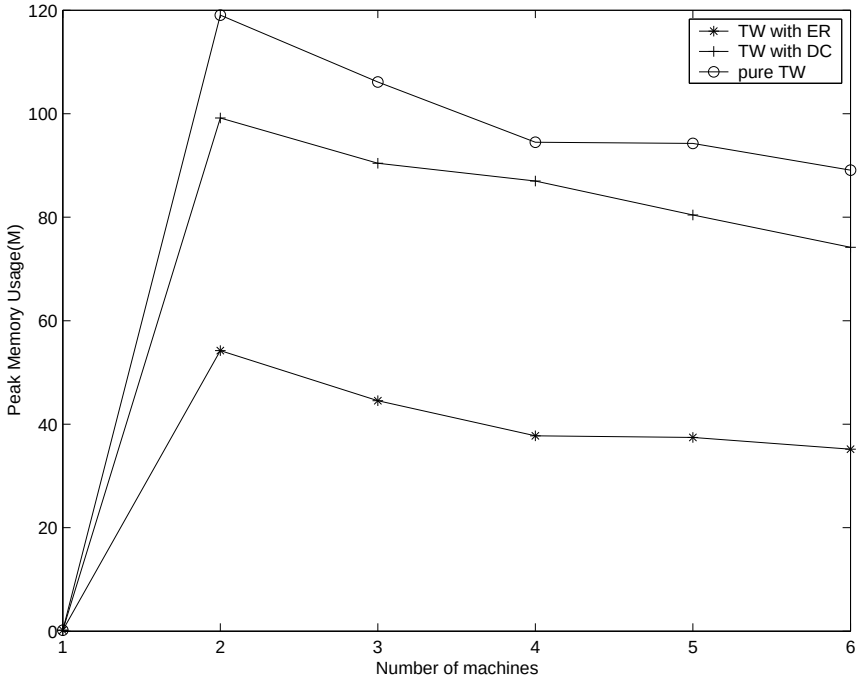


Figure 5–9: Memory consumption for S38584

5.5.2 Simulation Time

The simulation time vs. the number of machines for the 16 bit multiplier is presented in figure 5–10.

We observe from figure 5–10 that event reconstruction results in a 10% execution time improvement over dynamic checkpointing and a 40% improvement over Time Warp when 2 machines are used. The speedup decreases when more machines are used for the same reason that the improvement in memory consumption diminishes when more machines are used. The speedup obtained using event reconstruction is 3% better than dynamic checkpointing and 35% better than Time Warp when 6 machines are used.

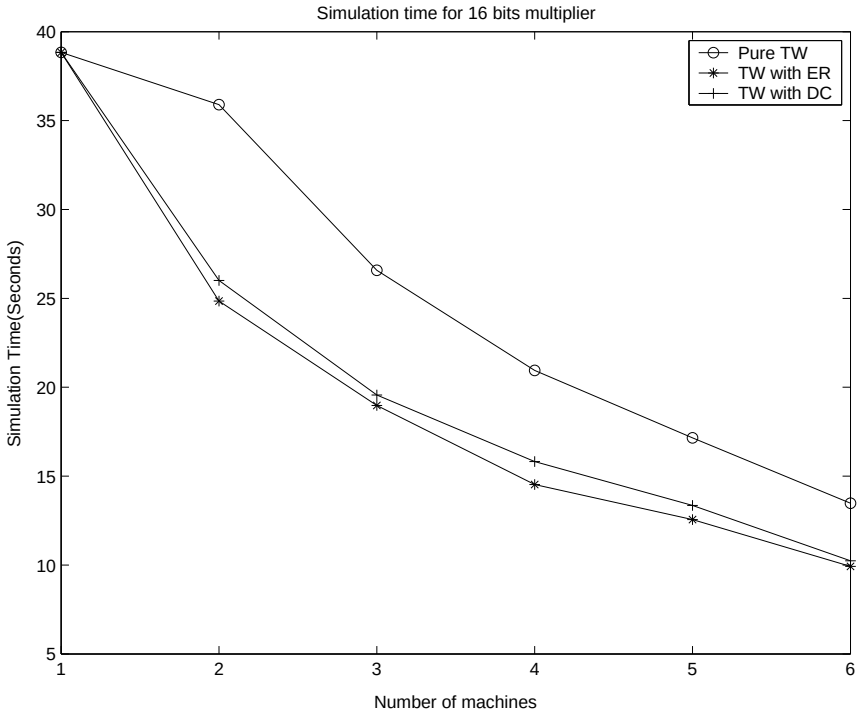


Figure 5–10: Simulation Time for 16 bits multiplier

Figure 5–11 presents the simulation time vs. the number of machines for S38584. The simulation time of pure Time Warp is 25.55 because of memory swapping. Both dynamic checkpointing and event reconstruction eliminate memory swapping. However, event reconstruction is 11% faster than dynamic checkpointing.

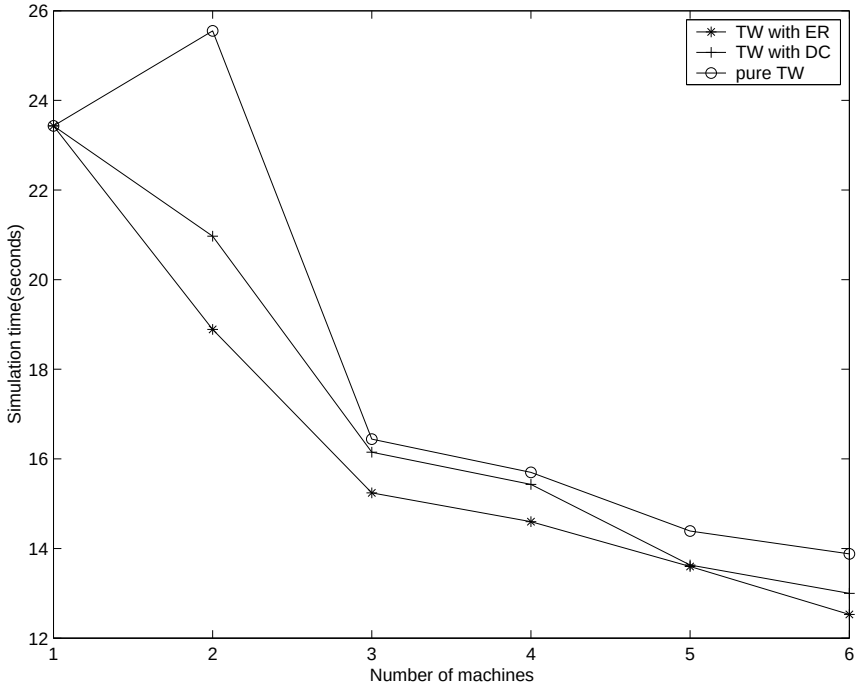


Figure 5–11: Simulation Time for S38584

CHAPTER 6

A multiway design-driven partitioning algorithm for distributed Verilog simulation

The research described in this chapter first appeared in [5].

6.1 Introduction

Modern VLSI systems are becoming increasingly complex, posing a never-ending challenge to sequential simulation. In order to accommodate the growing need for increased memory as well as the need for decreased simulation time, it is becoming increasingly necessary to make use of distributed simulation[7].

Time Warp[8] is an appealing technique for the distributed logic simulation of VLSI circuitry because it can potentially uncover a high degrees of parallelism in the VLSI system being simulated.

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

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

Most of the partitioning algorithms [49, 3, 74, 75, 76, 60, 4, 77, 78] for distributed/parallel VLSI simulation directly partition gate level netlists. These algorithms are typically used for floorplanning and placement, not for simulation. They can produce a big cutsize which is intolerable in a distributed VLSI simulation environment because of the communication costs which are a consequence of a large cutsize. Moreover, few partitioning algorithms take load balancing into account.

The ASIC design community has a well-established hierarchical design methodology. Every design is partitioned into blocks by functionality. The design hierarchy is reflected in modules and their instances in Verilog. 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 chapter is organized as follows. Section 6.3 is devoted to related research. In section 6.5, we introduce hierarchy in Verilog. Our distributed simulation environment DVS[1] could be found in chapter 4. In section 6.6, 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 htmis partitioning based on netlists is presented in section 6.7. The last section contains our conclusions and thoughts about future work.

6.2 Metrics of partitioning quality

Partitioning plays an important role in performance of parallel/distributed logic simulation [79, 80]. Since graph partitioning problem is NP-complete, most partitioning algorithms are heuristic algorithm. [65, 76, 60] shows three metrics determining the quality of partitioning, which are communication, load balance and concurrency. The goal of the partitioning algorithm is to minimize communication while achieving best load balance and maximum concurrency. Unfortunately the three metrics are sometimes contradictory. Therefore, the optimal partitioning algorithm tries to find the best tradeoff among these three factors.

6.2.1 Communication

Since our distributed simulation is executed on a network of workstations, communication is the most critical factor for the performance of the simulation. Furthermore, because of the known fine granularity of the computation in logic simulation, communication cost needs to be reduced as much as possible.

The amount of communication is typically estimated by the cutsize of the partitioning. The smaller the cutsize, the fewer messages are transferred between different partitions. In the experiment section of this chapter, the readers will see clearly that the cutsize strongly affects the simulation performance.

However, the traffic between different machines is not always proportional to the cutsize of the partitioning. In the actual simulation, some cut edges may be heavily loaded with communication traffic while other cut edges seldom carry any messages. Unfortunately most of the partitioning algorithm are static, which means the partitioning is done before the simulation. In order to obtain the dynamic traffic

information, it is necessary to use pre-simulation[81], which will be discussed later in this chapter.

6.2.2 load balancing

The performance of the distributed simulation is limited by its slowest machine. Therefore, in order to obtain the best simulation performance, the best strategy is to distribute the computing load evenly onto all computing machines and make them finish the computing task at the same timeframe.

In distributed Verilog simulation, we define the load of an LP to be the total number of events executed on the LP. However, since partitioning is done before simulation and it is not easy to obtain the number of events, we have to define load as a static metric. Usually we choose the number of gates in gate-level simulation since it is the rough estimate of the events in the LP. This simplification is based on assumption that all gates of the circuit are equally active during the simulation. As was the case for the communication, this is not always true in the actual simulation. Experimental results have shown that there are hot spots that are more active than others during the simulation. Again, pre-simulation is needed to evaluate the load balance.

Currently we assume our distributed simulation will be the dedicated users of the computing workstations. However, there could be external loads on the computing nodes. External load is not in the scope of our research about distributed Verilog simulation.

6.2.3 Concurrency

Concurrency is a metric which is easily confused with the concept of load balance, although there is major difference between these two metrics.

Load balance is dynamic measurement metric, although it is usually approached by static metrics such as the number of gates.

However, concurrency could be a good static metric. There is design level concurrency in a circuit. It means two subblocks could be simulated in parallel with little synchronization or even without synchronization. The design concurrency is the basis of parallel/distributed discrete event simulation for VLSI circuit simulation.

In [60] Kim defined a metric to measure concurrency. Two strategies of exploiting concurrency from a VLSI circuit are the following.

- Make use of the primary input or primary output. The primary input is a node with zero fan-in and primary output is a node with zero fan-out. String[46, 82] and cone partitioning[83] are such kind of partitioning algorithm. Its partition is generated by the fanout along with the primary input or output of the circuit.
- Make use of level sorting. All of the vertices at the same level are assigned to different partitions.

6.3 Related work

6.3.1 Non-iterative partitioning algorithm

A common approach to circuit partitioning consists of dividing the circuit into two or more blocks such that the number of connections between the blocks of the partition is minimized. The partitioning problem is NP-complete and commonly and, as a consequence, heuristic methods are used to achieve a solution. An excellent survey on the state of the art in VLSI circuit partitioning may be found in [84].

Random partitioning is the simplest method of partitioning. As the name implies, it assigns gates to processors randomly. It is a simple and efficient algorithm. However, it cannot produce the optimal cutsize.

String partitioning [46, 82] first distributes the primary inputs randomly onto the available processors because they are the first element to create new events during the simulation. The algorithm then continues in a depth-first manner by following one of several fanout branches until either an external output of the circuit or an already assigned unit is encountered. If there remain unassigned units, one of them is randomly chosen, placed into the partition with the least number of gates and the depth-first search is repeated until all gates belong to a partition.

Cone partitioning[49] is well suited for circuits that have primary inputs. Starting with the primary inputs of a circuit, all of the gates that are driven by the primary input are added to the output cone of that primary input. This procedure is repeated recursively for the added gates until a primary output is reached.

6.3.2 Iterative partitioning algorithm

In 1970, Kernighan and Lin(KL)[85] proposed a well-known heuristic for the two-way graph partitioning algorithm which has become the basis for most of the subsequent partitioning algorithms in this area. The algorithm is called an iterative improvement algorithm because it is based on the cell moves to improve the solution iteratively until a local minimum is obtained.

The KL algorithm[85] starts with an initial two-way partition. Then it performs a series of passes until a local minimum of cutsize is achieved. A pass consists of a number of pairwise cell swapping between the two partitions. Schweikert and

Kernighan [86] proposed a more practical model referred to as the hypergraph model for the circuit partitioning problem.

Fiduccia and Mattheyses [3] presented a modified version of algorithm KL[85] in order to speed up the search. They introduced a new data structure (bucket list of cell gains) to achieve linear run time per search. Moreover, they proposed a cell move strategy instead of swapping a pair of cells in one move, which allows more flexibility in selecting candidate cell to move.

Krishnamurthy [87] suggested that the lack of an intelligent tie-breaking scheme among many possible cell moves with the same gain could cause the FM algorithm to make bad choices. He enhanced the FM algorithm[3] with a look-ahead scheme that looks ahead up to r^{th} level of cell gains to choose a cell move.

Sanchis [88] extended FM algorithm[3] with Krishnamurthy's look-ahead scheme[87] to multi-way partitioning. Sanchis's algorithm is also the first hyper-graph multi-way partitioning algorithm since all the previous algorithm described are two-way partitioning algorithm. Sanchis's algorithm is extensively used as a benchmark in performance comparison for different multiway hypergraph partitioning algorithms.

All the FM-based[3] partitioning algorithms, such as KL[85], KR[87] and SA[88] algorithm are generally intuitive, flexible in adapting to different optimization objectives, easy to implement and relatively fast.

Park and Park[89] pointed out that the cell move operation is largely influenced by the balancing constraint. Therefore, they proposed a cost function that comprises both the cutsize and the balance degree that is the sum of all size differences between different partitions. The balance degree is associated with a positive weighting factor. They proved that a minimum cost multiway partitioning obtained by their algorithm

corresponds to a balanced minimum cutsize as defined in SA algorithm[88] if the weighting factor is larger than the number of cells in a circuit. The SA algorithm[88] is then used to solve the multi-way partition problem under their objective function.

Dutt and Deng[4] observed that the FM-based algorithms could only remove small clusters from the cutset while it may lock bigger clusters in the cutset. They divided the cell gain into initial gain calculated before a cell movement and the update gain generated from the cell movement afterwards. By focusing on the update gain when choosing cells to move, they reported very successful results for bipartitioning experiments.

Harypis [6] introduces a coarsening phase in a multilevel hypergraph partitioning algorithm hMetis. During the coarsening phase, a sequence of successively smaller hypergraphs is constructed. The purpose of coarsening is to create a smaller hypergraph while preserving the partitioning quality obtained from the original hypergraph. The authors claim that hmetis produces partitions that are consistently better than other widely used algorithms and is one to two orders of magnitude faster than other algorithms.

Dasdan and Aykanat[90] developed two multiway partitioning algorithms using a relaxed locking mechanism. The first one (PLM) uses the locking mechanism in a relaxed manner. It allows multiple moves for each cell in a pass by introducing the phase concept so that each pass may contain more than one phase and each cell has a chance to be moved only once in each phase. The second algorithm (PFM) does not use the locking mechanism at all. A cell can be moved as many times as possible per pass based on its mobility value. The performance of the two algorithms was compared with the Simulated Annealing algorithm[91] and Sanchis's algorithm

[88] on some benchmark circuits. The results of the algorithms outperform Sanchis's algorithm significantly on multiway partitioning. The performance of the algorithms are comparable to Simulated Annealing algorithm[91] while the algorithm is much more efficient.

Cong and Lim[92] proposed a multiway partitioning algorithm with pairwise cell movements. It starts with an initial multiway partition and then applies the bipartitioning heuristic (FM algorithm[3]) to pairs of blocks concurrently to improve the quality of the overall multiway partitioning solution.

Yang and Wong[93] presented a network flow based partitioning algorithm to solve bipartitioning problem and they claimed that multiway partitioning can be accomplished by recursively applying the network flow based algorithm.

6.3.3 Iterative partitioning algorithm utilizing design hierarchy

It is worthwhile noting that the CLIP[4] and hMetis[6] algorithm tries to detect and restore the cluster destroyed by the iterative partitioning algorithm based on the flattened netlist.

[4] and [6] try to reduce the size of the hypergraph from the bottom up, i.e. they extract clusters from the flattened netlist without worsening the quality of the partitioning of the original netlist. Our algorithm works from top-to-bottom-it flattens the design hierarchy step by step and compromises between the load balancing constraint and the minimum cutsize.

Iterative algorithms generally work on any hypergraph while our algorithm specifically targets distributed Verilog simulation. The main purpose of our algorithm is to try to keep the Verilog instance (actually the design hierarchy) intact from

the beginning. It is much easier than restoring it from the debris produced by first flattening the netlist. Moreover, the quality of the resulting partition should be better than the cluster restoration and hypergraph coarsening.

Algorithms which try to take advantage of the information included in the design hierarchy include the following.

Chau-Shen [94] proposes an architecture driven partitioning algorithm for netlists with multiterminal nets. The target architecture was a multifield-programmable gate array (FPGA). The goals of the algorithm are to minimize the number of FPGA chips used and to maximize routability.

Tun [68] uses a module-based simulation mapping method. Although the details of the algorithm are not described in the paper, the author states that it reduces the communication cost and achieves a better load balancing.

K.H. Chang[95] uses the module tree as the data structure instead of the circuit hypergraph. Modules are not moved by the algorithm. Nor does it use an iterative improvement technique. The author does not mention the cutsize achieved by the algorithm and concludes that the algorithm achieves better performance than a sequential simulation and is efficient.

Jong-Sheng [96] proposes a module migration based partitioning algorithm which tends to keep the cluster intact in order to reduce the net cut size. The algorithm implicitly promotes the move of clusters of modules during the module migration process by paying more attention to the neighbours of moved modules, relaxing the size constraints temporarily during the migration process, and controlling the module migration direction. Load balancing was not considered in this algorithm.

Iterative algorithms start from an initial partitioning and try to improve it. The well-known iterative algorithms for circuit partitioning are CLIP/CDIP[4], Metis/hMetis[6] and F-M[3]. It is worthwhile noting that the CLIP[4] algorithm tries to detect and restore the cluster destroyed by the iterative partitioning algorithm based on the flattened netlist.

Harypis[6] introduces a coarsening phase in a multilevel hypergraph partitioning algorithm. During the coarsening phase, a sequence of successively smaller hypergraphs is constructed. The purpose of coarsening is to create a smaller hypergraph while preserving the partitioning quality obtained from the original hypergraph. The authors claim that hmetis produces partitions that are consistently better than other widely used algorithms and is one to two orders of magnitude faster than other algorithms.

Dutt and Deng [4] and [6] try to reduce the size of the hypergraph from the bottom up, i.e. they extract clusters from the flattened netlist without worsening the quality of the partitioning of the original netlist. Our algorithm works from top-to-bottom-it flattens the design hierarchy step by step and compromises between the load balancing constraint and the minimum cutsize.

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

6.4 Motivation and objective

The existing partitioning algorithms are usually performed on the flat netlist of the circuit. The purpose of the partitioning algorithms are for circuit layout. Currently there is no partitioning algorithm specifically designed for distributed Verilog simulation while Verilog simulation has its own characteristics in terms of partitioning quality.

First of all, netlist described by Verilog contains design hierarchy that has multiple Verilog instances. The Verilog instances are the best potential candidate for the initial partition since the Verilog instances are representation of the design blocks that are supposed to be coupled loosely. The existing partitioning algorithm needs to restore clusters in the flat netlist through computation of the strong connected component in the circuit hypergraph.

Second, the partition quality of the existing partitioning algorithms are poor. The performance of distributed Verilog simulation will be worse than the sequential Verilog simulation.

Third, the efficiency of the existing partitioning algorithm is relatively low. There are some efforts to reduce the number of the hypergraph nodes. However, the partitioning of the circuit with millions of gates still takes long time.

In order to enhance the solution quality for the iterative partitioning algorithm, we need to take advantage of the design hierarchy in netlist described by Verilog in order to overcome the above disadvantages of the existing partitioning algorithm. This is the motive for developing our new algorithm.

The primary objectives of our partitioning algorithms are as follows.

1. The partitioning algorithm should produce the minimal communication and make workload among different computers as balanced as possible.

2. The partitioning algorithm should preserve the design hierarchy by the hardware design engineers
3. The partitioning algorithm should be able to choose the optimal number of computing nodes
4. The partitioning algorithm should take advantage of the primary input such as flip-flop or registers to exploit the concurrency in the design.

6.5 Hierarchy in Verilog

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

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

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

Figure 6–1 shows a design hierarchy described by Verilog. The left side of the figure is the Verilog source code while the right side displays the design hierarchy and its interconnection. Coupling is usually loose between Verilog instances and is tight inside a Verilog instance (at least for a good VLSI design). Therefore, if the circuit is cut at instance boundaries, the cutsize will be smaller and inter-processor communication will be reduced.

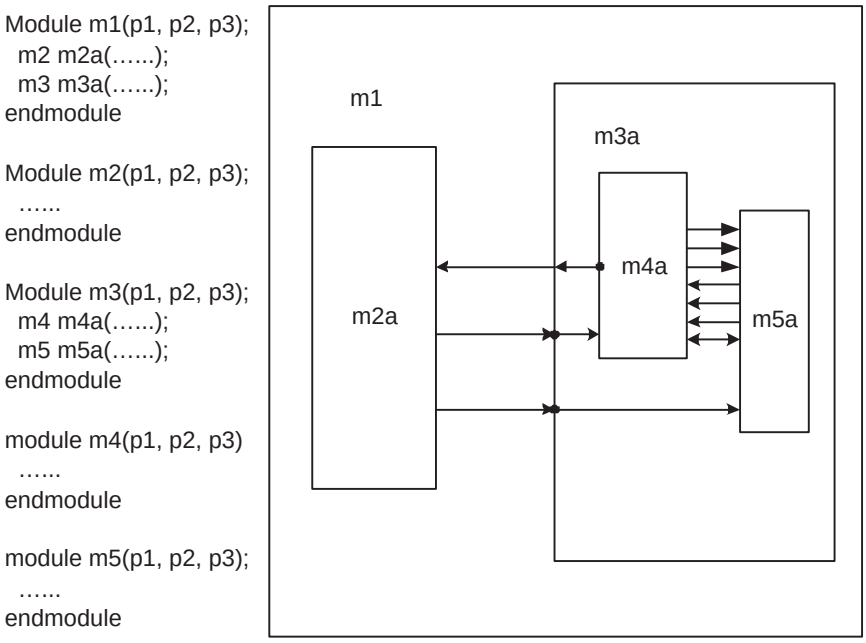


Figure 6–1: Verilog module/instances and interconnection

We should note that not only does 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 done before elaboration we are able to take advantage of the design information.

6.6 Implementation

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

6.6.1 hypergraph

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

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

Figure 6–2 contains a hypergraph which is composed of two kinds of vertices, gates and super-gates (Verilog instances).

In figure 6–2, there are two Verilog instances, u1 and u2 which are represented by two vertices in the hypergraph. However, in the zoom-out eclipse we see that both u1 and u2 have their own sub-graphs, each of which include multiple gates or Verilog instances.

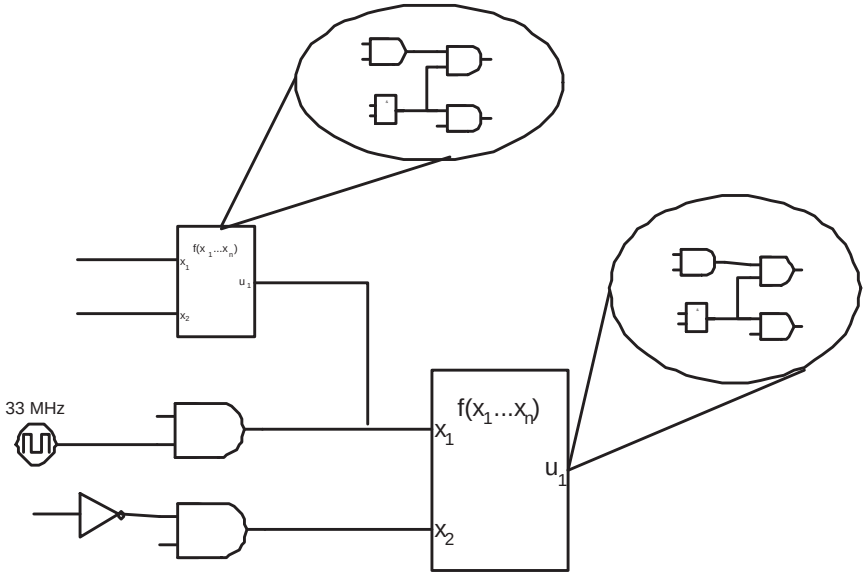


Figure 6–2: Hypergraph represented by Verilog

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

Table 6–1: Logic values and their purposes

Before we introduce the data structure used in the algorithm, we define two properties of a vertex. We say that a vertex is not visible if it is inside of a Verilog instance , otherwise it is visible. We say that a vertex is primitive if it cannot be decomposed into multiple vertices, otherwise it is not primitive. Consequently there are four kinds of vertices, as shown in table 6–1.

For example, in figure 6–2, all of the nodes inside the zoomout ellipse are of kind C, while the node zoomed out is of kind B. The properties of the vertex can change during the partitioning process. For example, the vertex inside of a Verilog instance

will become visible after flattening. Any invisible vertex will have the same partition id as its parent. Therefore, only visible vertices will appear in the hypergraph.

The complexity of any partitioning algorithm is proportional to the number of vertices. A reduction in the number of vertices in a hypergraph results in simpler hypergraph and a more efficient partitioning algorithm.

6.6.2 data structure

Figure 6–3 shows the data structure used in the partitioning algorithm. The hypergraph is represented as a vertex vector and an edge vector. Each vertex contains the load, a pointer to its parent, the partition id, the neighbouring vertices list, the Behring edges list and the input ports list. The input ports list contains all of the input ports of the vertex and the internal vertices connected to the input ports while the output ports list contains all of the vertices to which it connects. The ports can be used to flatten a vertex. All of the invisible vertices are expanded into visible vertices when a vertex is flattened. Details of flattening are explained in subsection 6.6.8.

The bucket is the data structure used to arrange vertices in the order of their gain values. It was first used in [3] in order to improve the runtime performance of the FM[3] algorithm. The bucket data structure is inspired by the bucket sorting algorithm. It has the following two advantages.

- Locating a vertex with the highest gain in the bucket is constant time
- After gain updating, the re-insertion of a vertex into the bucket is accomplished in constant time

As shown in figure 6–4, 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

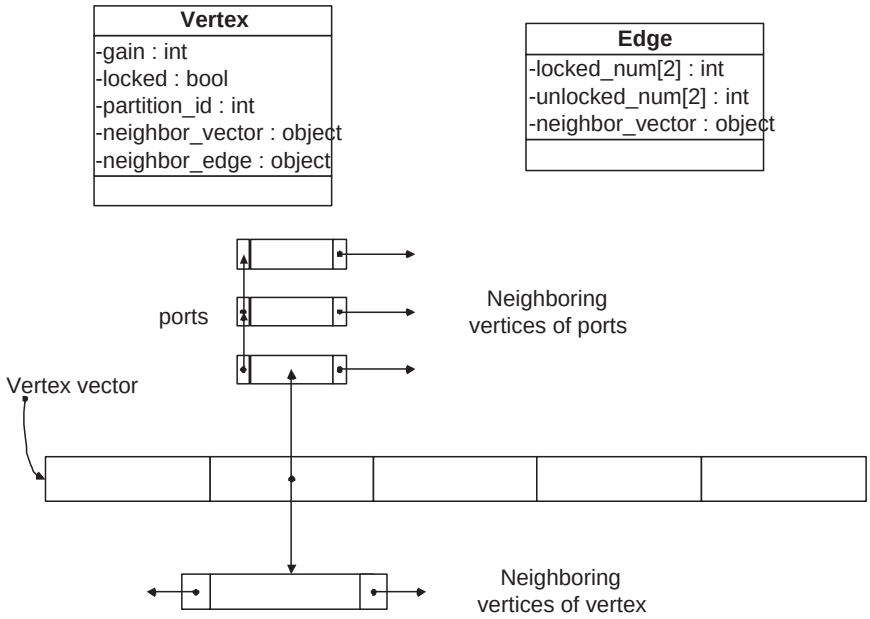


Figure 6–3: Data structure of the partitioning algorithm

the gain value. All vertices on the same row form a double 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[73] show that the runtime performance of FM[3] based on single linked list could be 300 times slower than the algorithm based on double linked list.

6.6.3 Verilog parser and hypergraph builder

The Verilog parser reads in the Verilog source code and builds the hypergraph. In the hypergraph, the Verilog instances are treated as super-gates and are therefore represented as one vertex. The Verilog parser is extended from the original Verilog parser of Icarus[58] Verilog simulator.

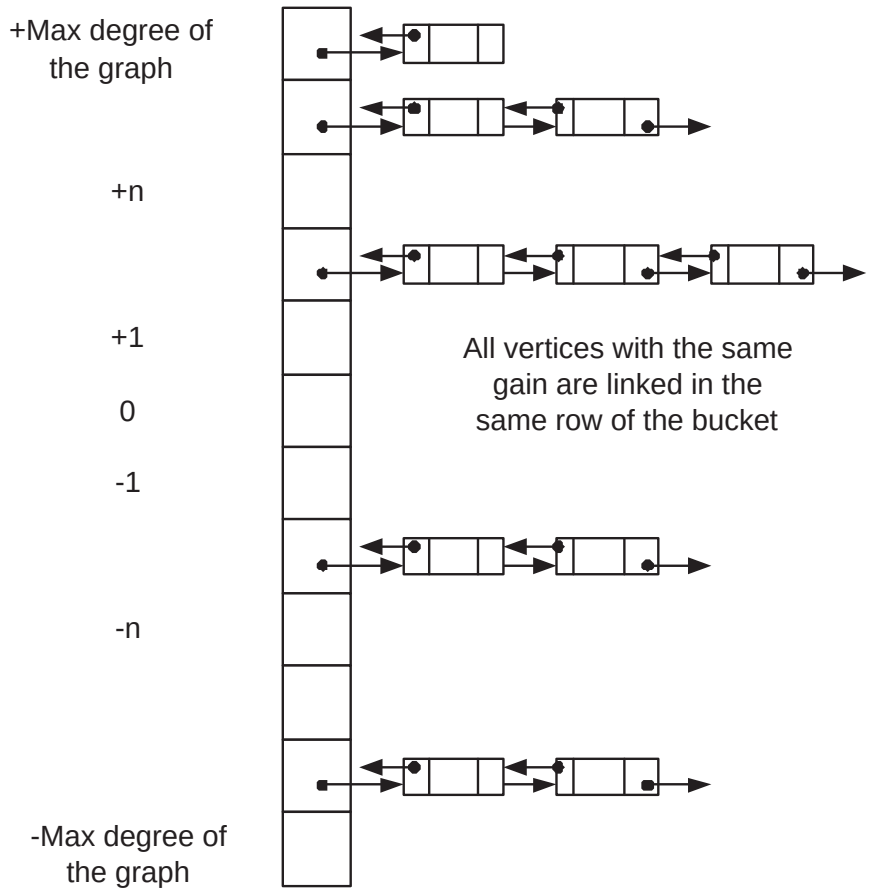


Figure 6–4: Bucket data structure for vertex movement

6.6.4 Cutsize and gain from the movement

A circuit netlist is modelled by a hypergraph G=$< V, E >$ 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 while r is the number of the partitions in which the cut resides. The cutset of the circuit consists of all the edges which are cut.

The cutsize of the circuit is defined to be the sum of cost of all cuts in the cutset as shown in formula 6.1. In the formula, c_i stands for the i_{th} cut in the circuit while n stands for the number of cuts in the circuit.

$$cutsize = \sum_{i=1}^{n} cost(c_i) \tag{6.1}$$

Gain of the vertex movement is defined to be the immediate reduction in cutsize as shown in formula 6.2. In the formula, m is the number of cuts after the vertex movement while n is the number of cuts before the vertex movement. c_i stands for the i_{th} cut in the circuit. The negative gain means there is no reduction in the cutsize.

$$gain = \sum_{i=1}^{m} cost(c_i) - \sum_{i=1}^{n} cost(c_i) \tag{6.2}$$

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

6.6.5 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.

$$load * (1/k - b/100) <= load[i] <= load * (1/k + b/100) \quad (6.3)$$

In the formula 6.3, 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*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 6.7.

6.6.6 Initial partitioning

Our initial partitioning algorithm is an improved depth-first-search partitioning algorithm whose pseudo is shown in figure 6–5. The algorithm traverses the hypergraph from the primary inputs and adds vertices into a partition. The initial partitioning terminates when all of the primary input ports are visited.

The partitioning algorithm could preserve concurrency in the circuit since it distributes the primary inputs into different partitions, as shown in figure 6–6.

6.6.7 Iterative moving

The iterative moving of hypergraph nodes is the same as in the Fiduccia-Mattheyses (FM) [3] algorithm. It modifies the initial partition by a sequence of moves which are organized into passes. At the beginning of a pass, all of the vertices are free to move (they are unlocked), and each possible move is labelled with the immediate change in the total cost which it would cause; this is called the gain of

```
initial_partitioning(G, PI, k)
  /*
    G is the circuit graph, PI is the set of primary input and k is the number of
desired partitions
    L[] is the array of lists for the graph traversal. Both insertion and removal
operation are done at the head of the list
    VISITED is the array to indicate whether the vertex is visited or not. The
initial value of the array is 'false'.
  */
 //distribute primary input evenly to all partitions
 for all vertices v in the set of PI
 {
   p = (index of v in PI) * k / (sizeof(PI)); //p is the partition number for vertex v
   insert(L[p], v);
 }

 //partition the circuit graph into disjoint sub-graphs
 for (i=0;i<k;i++)
 {
  while (L[i] is not empty)
  {
   v = remove(L[i]);
   assign v to partition i;
   VISITED[v] = true;

   for all vertices w adjacent to v such that w is not visited
   {
    insert(L[i], w);
   }
  }
 }
```

Figure 6–5: Pseudo code of the initial partitioning algorithm

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

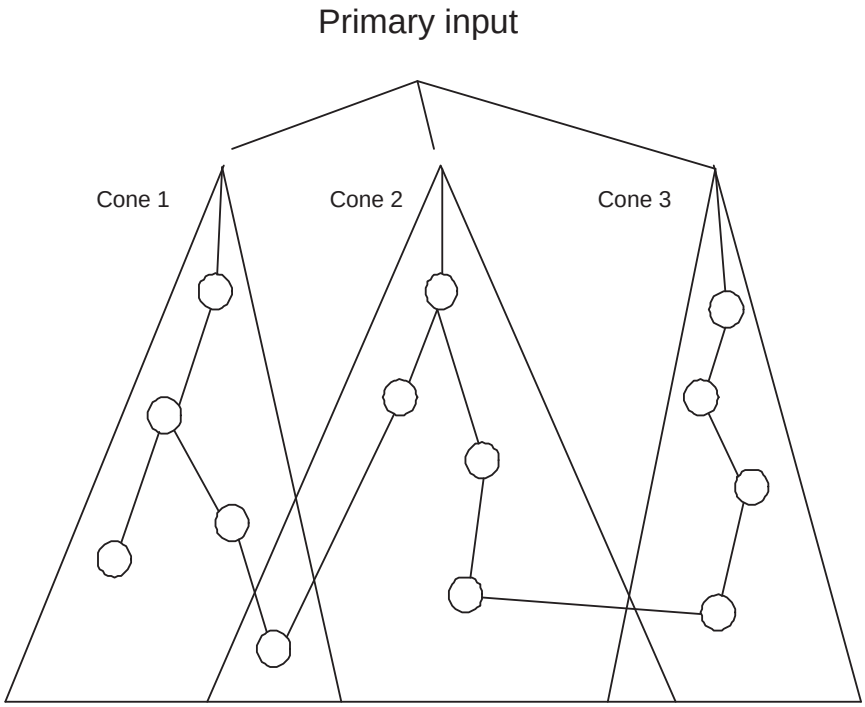


Figure 6–6: Initial partitioning result

terminates when a pass fails to improve the quality of the solution. The whole process of the iterative moving is shown in figure 6–7. The detail explanation of the iterative moving steps follows.

- Step 1: Calculate initial gains for all vertices
- Step 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.
- Step 3: Locate base vertex from either bucket
- Step 4: Move the selected base vertex
- Step 5: Update gains of the neighboring vertices of the base vertex

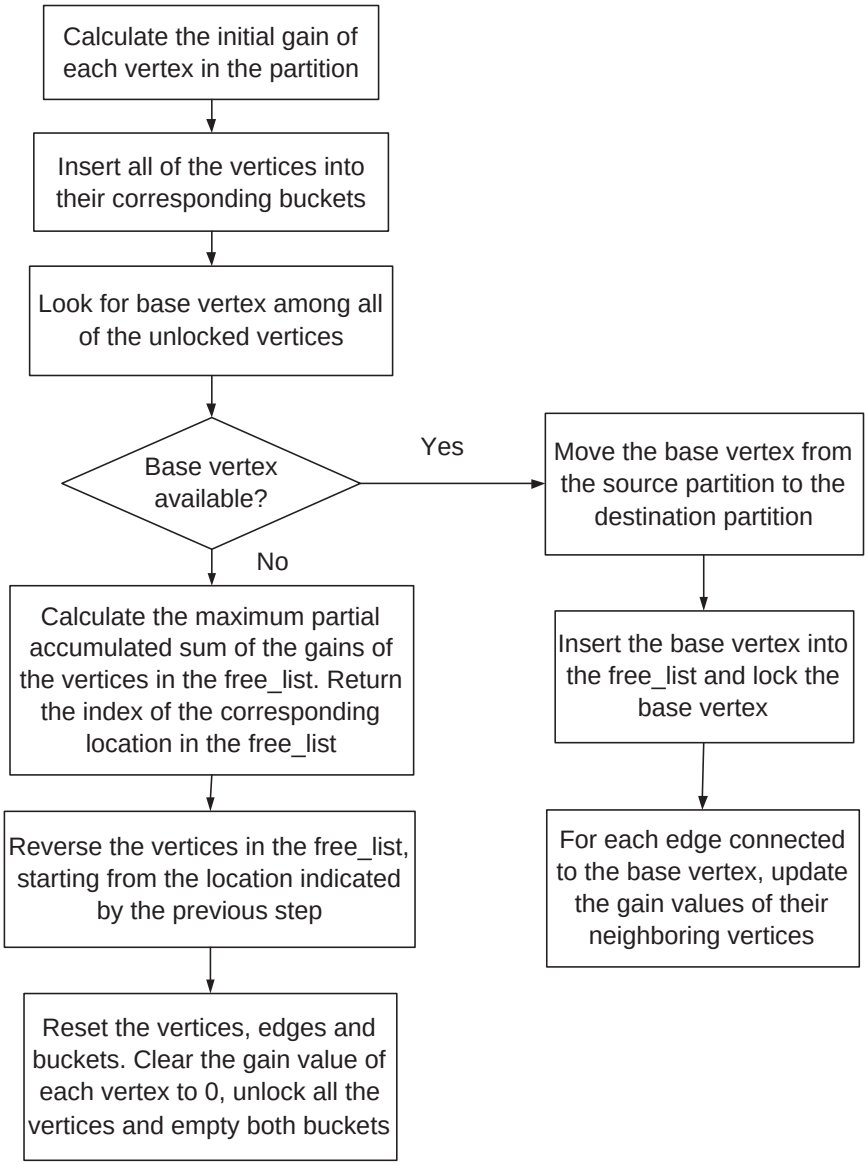


Figure 6–7: The iterative moving of vertices

- Step 6: Calculate the maximum partial accumulated sum of gains for the current pass
- Step 7: Reverse selected vertices

6.6.8 Flattening

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

Figure 6–8 shows the original hypergraph and the result of the flattening.

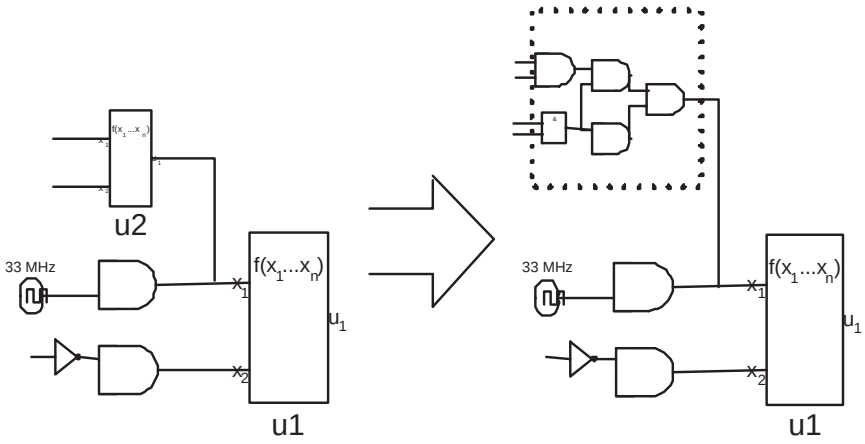


Figure 6–8: Flattening of the circuit hypergraph

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 re-distribute the load after the flattening.

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

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

We observe that cutsize will 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 trigger a rollback explosion. Details are presented in section 6.7.

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

6.6.9 Tie breaking

Tie-breaking strategies plan an important role in circuit partitioning since different tie-breaking scheme could lead to different local minima of the cutsize. Our algorithm uses affinity to break tie between different vertex candidates to move. The affinity is defined to describe how close a vertex is bound to its parent partition. At the beginning

of the algorithm, the affinity of the vertex is its level from the root of the hypergraph. It means the leaf vertices have the smallest affinity with its parent partition.

The idea of affinity extends from the tie-breaking strategy used by CLIP[4], which is to give high priority to those neighbors of the moving cells in the next round of moving based on the locality principle. CLIP[4] algorithm takes this approach in order to remove large cluster from the cutset. The authors of CLIP observed that large cluster of vertices are still trapped in the cutset despite that the small cluster of vertices could be extracted out of the cutset. Sub-clusters which are part of the larger cluster are able to move between the cutline. However, while one sub-cluster moves in one direction, another may move in the opposite direction later. This movement will finally be stabilized and stopped with the sub-clusters residing on both sides of the cutline.

The basic idea of CLIP[4] is that all the neighboring vertices of the moved vertex should be given higher priority to be moved. The rational of the approach is that if one vertex of a large cluster is moved out of the cutset, all of the other vertices of the same cluster should be moved in the same direction in order to move the whole cluster out of the cutset eventually.

6.6.10 Pairwise multiway partitioning algorithm

The existing multiway partitioning algorithm can be classified into two primary approaches: recursive and direct.

The recursive approach applies bipartitioning recursively until the desired number of partitions is obtained while the direct approach partitions the circuit directly. Among all the previous algorithm mentioned in the related work section, Relaxed locking[90] and pairwise partitioning[92] belongs to direct multiway partitioning algorithms. Figure

6–9 shows the recursive partitioning algorithm while figure 6–10 shows the pairwise partitioning algorithm.

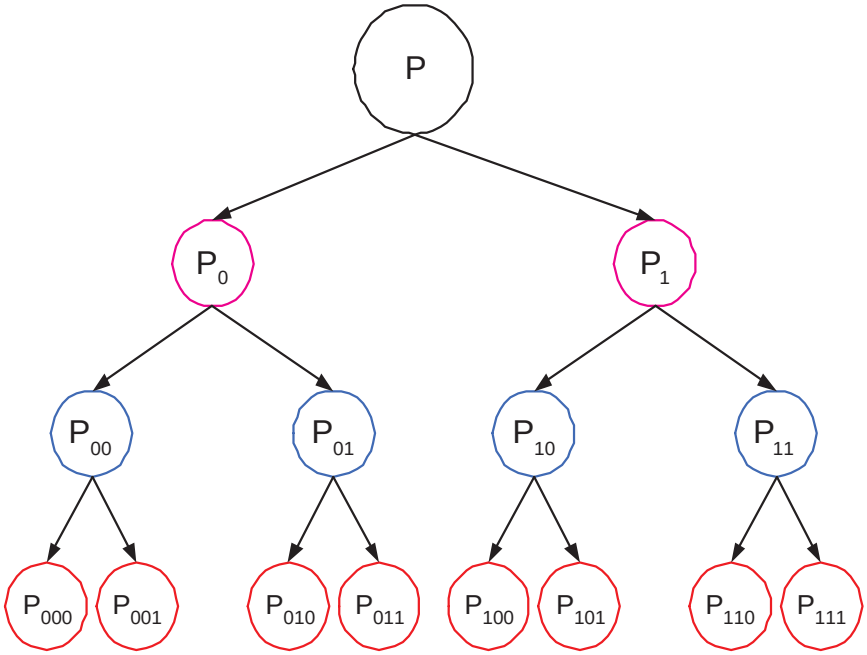


Figure 6–9: Recursive multiway partitioning algorithm

Pairwise partitioning is a direct multiway partitioning algorithm. In pairwise multiway partitioning, the initial partitioning partitions the circuit into k partitions instead of just 2 partitions as in the recursive partitioning algorithm. In the next step, the algorithm chooses two partitions based on some criteria. Then swapping of circuit elements is executed recursively between the paired partitions in order to further minimize the cut-size of the pair. The pairing and the recursive moving is done iteratively until the cut-size is minimized and the load balancing constraint is achieved.

Figure 6–10 shows the principle of the pairwise partitioning algorithm. In the initial partitioning, we can see that the circuit is partitioned into 8 partitions

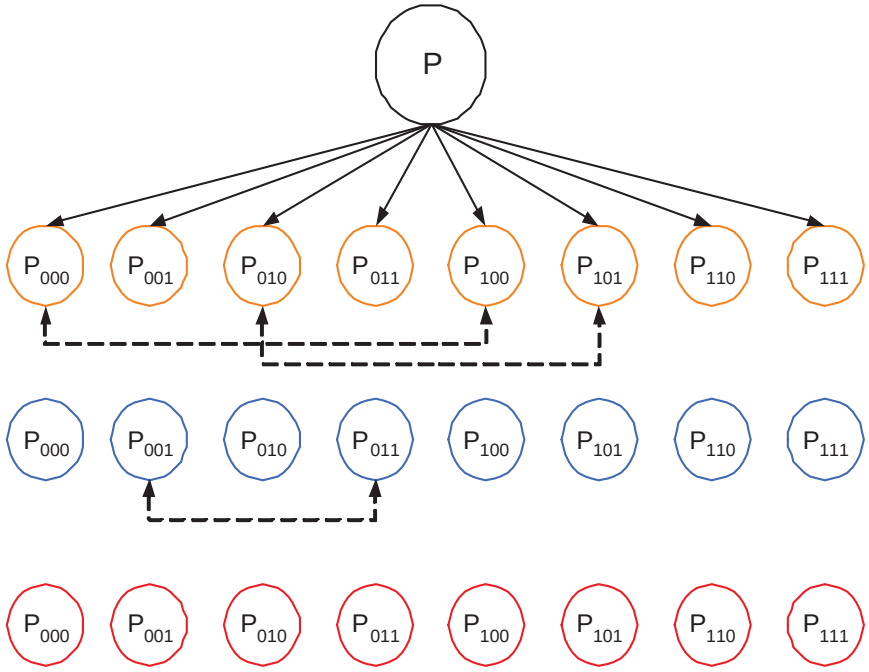


Figure 6–10: Pairwise multiway partitioning algorithm

directly. Then in the next step, P000 and P100 are paired together for iterative cut-size improvement as indicated by the dotted line. Later P001 and P011 are paired together.

There are a number of criteria which can be used to chose the pairs of partitions.

- Random

  The pairing of partitions is random. It is simple and efficient, but the pairing quality is not good

- Exhaustive

  The pairing of partitions will be every combination of the partitions. It is computationally complex but produces better results because it is able to climb out of local minima.

- Cut-based

The pairing is done between the two partitions between which the cut-size is maximum.

- Gain-based

  The pairing is done between the two partitions between which the cut-size reduction is maximum.

The recursive algorithm is computationally simple and fast. However, it suffers from several limitations. If the number of partitions are not a power of 2, the desired number of multiway partition cannot be achieved. Furthermore, as the algorithm proceeds, it becomes harder to reduce the cut-size since the partitioning is performed on finer and finer hypergraphs. This observation, combined with the assertion in [92] that their k-way pairwise partitioning algorithm produces good results efficiently led us to chose the direct algorithm

6.6.11 Apply pre-simulation to find the optimum partitioning

From the previous description of three metrics for measuring partitioning quality in section 6.2, we mentioned both communication and load balance are dependent on presimulation[81].

Pre-simulation[81] is an efficient approach for evaluating the quality of a partition. [81] provides evidence 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 speedup during the presimulation. The partition

which produces the the best speedup for some choice of k and b is used in the circuit simulation.

We used brute force pre-simulation strategy which runs all combinations of parameters k and b.

6.6.12 Putting it all together

Figure 6–11 contains a flowchart of the algorithm. After the initial cone partitioning, the pairing process is executed in order to pick candidates for iterative movement. Then the algorithm 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 to do incremental flattening as discussed in section 6.6.8. The pairing, iterative movement and flattening process are repeated until there is no pairing configuration is available. At the end of the partitioning algorithm, the minimum cutsize is achieved and load balancing constraint is met as well.

The termination criteria of pairwise run is based on gain of cutsize between all possible combination of pairing to ensure the convergence of the partitioning algorithm. If all pairing configurations cannot achieve the cutsize gain and improve the partition, the partitioner stops.

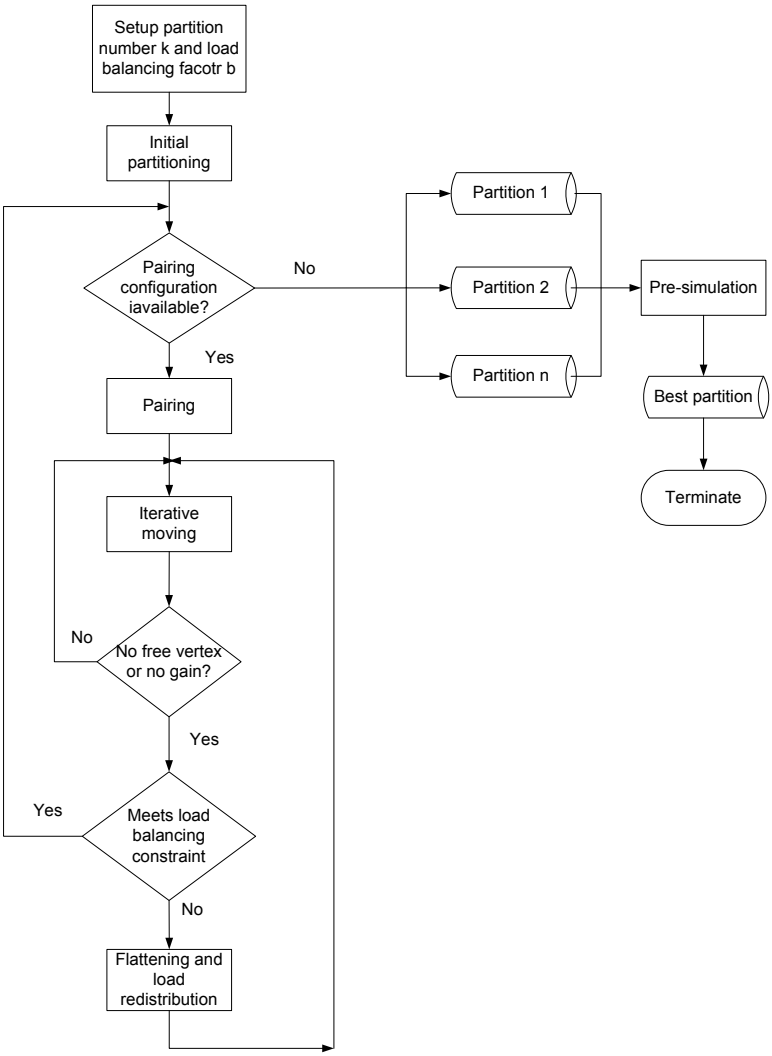


Figure 6–11: Flowchart of the design-driven partitioning algorithm

6.7 Experiments

All of our experiments were conducted on a network of 4 computers, each of which has AMD Athlon (CPU 1G) processors and 512M RAM. They are interconnected by a 1Gbit Ethernet network. All of the machines run the Linux operating system while MPICH-GM is used for message passing between different processors.

We used the synthesized netlist of 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 also used ISCAS'89 benchmark circuit suite to prove our design-driven partitioning algorithm could also work well in the pure flat netlist without the auxiliary design hierarchy information.

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 without partitioning.

In the experiments, we compare the performance of DVS with the design-driven partitioning algorithm with that of DVS using htmis[6] as the partitioner.

6.7.1 Cutsize for Viterbi decoder

We use different values of k and b to generate different cutsizes. The hyperedge cutsize is defined as the number of hyperedges that span multiple partitions. Table 6–2 shows the hyperedge cutsize produced by our design driven iterative partitioning algorithm while table 6–3 lists the cutsize produced by the hMetis partitioning algorithm. The parameter b is the load balancing factor defined in formula 6.3 while k is the number of partitions.

Table 6–2 and table 6–3 reveal that our algorithm resulted in a significantly smaller cutsize than the one produced by hMetis. But from the experiment result we also know one of the drawbacks of our algorithm. Our algorithm is more sensitive to the load balancing factor. When the load balancing constraint is strict, the algorithm could produce almost the same cutsize as hMetis partitioning algorithm. We attribute

k	*b*	*Hyperedge cut*
2	2.5	2428
2	5	1827
2	7.5	905
2	10	633
2	12.5	598
2	15	513
3	2.5	2930
3	5	2227
3	7.5	1230
3	10	894
3	12.5	863
3	15	790
4	2.5	3230
4	5	2326
4	7.5	1433
4	10	979
4	12.5	935
4	15	887

Table 6–2: cutsize with design-driven partitioning algorithm

the sensitivity of load balancing to the load imbalance of the design hierarchy. Since there is more Verilog instance flattened, it means the algorithm will downgrade more to the hMetis partitioning algorithm.

6.7.2 Cutsize for ISCAS benchmark circuit

In order to prove that our multiway design-driven iterative partitioning algorithm also works in the flattened netlist, we conduct experiments on the ISCAS'89 benchmark circuit. The experiments show that our design-driven iterative partitioning algorithm is downgraded to a normal FM partitioning algorithm on the flattened netlist without any auxiliary design hierarchy information.

k	*b*	*Hyperedge cut*
2	2.5	2675
2	5	2673
2	7.5	2673
2	10	2669
2	12.5	2668
2	15	2665
3	2.5	2932
3	5	2932
3	7.5	2931
3	10	2935
3	12.5	2931
3	15	2927
4	2.5	3195
4	5	3195
4	7.5	3191
4	10	3191
4	12.5	3191
4	15	3191

Table 6–3: cutsize with hmetis partitioning algorithm

The table 6–4 and 6–5 shows the cutsize generated by the design-driven partitioning algorithm and FM partitioning algorithm on ISCAS'85 benchmark circuit, s35932 and s38584 that are described in Verilog. The s35932 has 12204 gates, 3861 inverters and 1728 D-type flip-flops. The s38584 has 11448 gates, 7805 inverters and 1452 D-type flip-flops. Please note DDP as the abbreviation of the design-driven partitioning algorithm.

When the algorithm works on the flattened netlist, we found that the sensitivity to the load balancing factor is much less compared to the netlist with design hierarchy. But unfortunately the cutsize produced by the design-driven partitioning algorithm is exactly same as the FM partitioning algorithm. However, this is what we expect when

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

Table 6–4: cutsize on ISCAS benchmark circuit s39592

we design the algorithm. When there is no design hierarchy information available, the algorithm will downgrade to the normal FM partitioning algorithm.

6.7.3 Presimulation

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 seconds.

Table 6–6 shows the simulation time and speedup with these combinations. We list the best partitions as determined by the largest speedup in table 6–7. Please note that all of the partitions produced by hMetis are slower than the sequential simulation.

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

Table 6–5: cutsize on ISCAS benchmark circuit s38584

We attribute this to the huge communication cost. The communication cost leads to enormous rollbacks and slow GVT computation. Both of the factors leads to huge memory consumption. Since memory swapping happens so frequently, we cannot expect any speedup over the sequential Verilog simulation.

6.7.4 Simulation time

Table 6–8 and figure 6–12 shows the simulation times and speedups with different combinations of the load balancing factor and cutsize. The sequential simulation time of the circuit is 3639.70.

From table 6–8, we know that the minimum cutsize does not always result in the best performance since the performance is also dependent on load balancing.

k	*b*	*Cutsize*	*Simulation time (Seconds)*	*Speedup*
2	2.5	2428	61.79	0.62
2	5	1827	41.86	0.93
2	7.5	905	30.65	1.27
2	10	633	25.78	1.51
2	12.5	598	23.59	1.65
2	15	513	29.72	1.31
3	2.5	2930	56.42	0.69
3	5	2227	39.72	0.98
3	7.5	1230	28.87	1.35
3	10	894	21.50	1.81
3	12.5	863	22.37	1.74
3	15	790	25.44	1.53
4	2.5	3230	88.47	0.44
4	5	2326	42.78	0.91
4	7.5	1433	19.86	1.96
4	10	979	24.80	1.57
4	12.5	935	21.04	1.85
4	15	887	24.18	1.61

Table 6–6: Pre-Simulation time with design-driven partitioning algorithm

k	*b*	*Cutsize*	*Simulation time (Seconds)*	*Speedup*
2	12.5	598	23.59	1.65
3	10	894	21.50	1.81
4	7.5	1463	19.86	1.96

Table 6–7: Best partition produced by design-driven partitioning algorithm

We got 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 6–8, we also observed that the load balancing becomes more and more important as the number of

k	*b*	*Cutsize*	*Simulation time (Seconds)*	*Speedup*
2	12.5	598	2201.98	1.65
3	10	894	2033.35	1.79
4	7.5	1463	1905.60	1.91

Table 6–8: Simulation time with design-driven partitioning algorithm

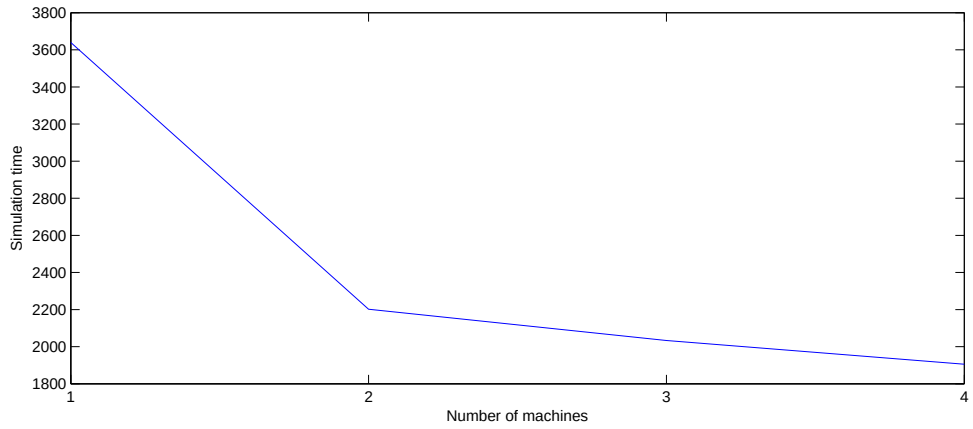


Figure 6–12: Simulation time

machines increases from 2 to 4. Because of increasing cutsize, we didn't 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 6–8.

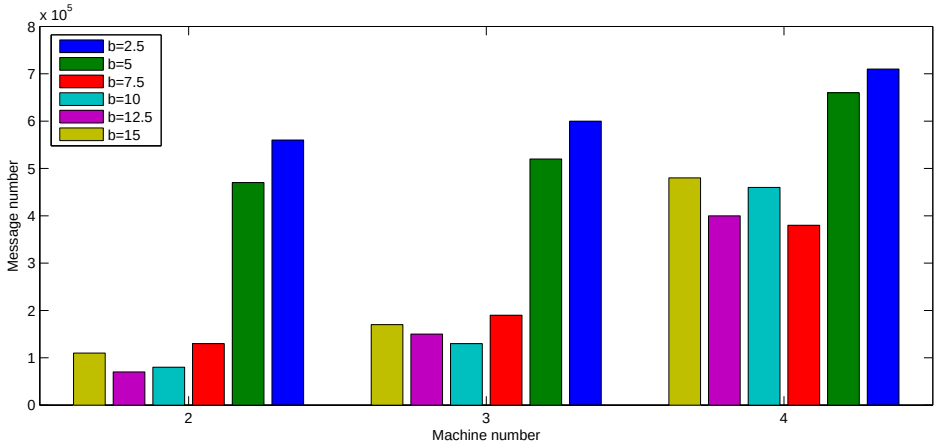


Figure 6–13: message number during the pre-simulation

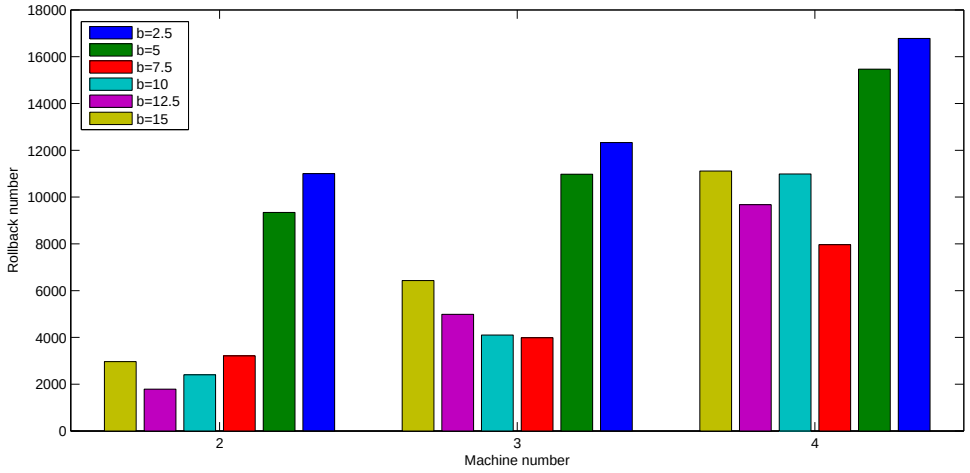


Figure 6–14: rollback number during the pre-simulation

6.7.5 Messages and Rollback

Figures 6–13 and 6–14 confirm the relationship between load balance and communication. Relaxing the load balancing constraint (i.e. increasing b) results in fewer messages and rollbacks. With an increase in the number of machines,

communication increases and both of these quantities increase. These results underscore the importance of pre-simulation in picking the final partition.

6.8 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 cut size because they operate on a flattened netlist. Our design-driven partitioning algorithm yields a significant reduction in cutsize compared to such algorithms by taking advantage of hierarchical design information. Moreover, it preserves the locality expressed in Verilog modules and instances. The algorithm produces a 4.5 fold reduction in cutsize compared to the hmetis [6] partitioning algorithm. The reduction in cut size and the preservation of locality lead to a speedup of 1.91 on four machines than the sequential simulation.

CHAPTER 7

Conclusions and future directions

The value of the distributed Verilog simulator mostly depends on how well it can achieve simulation speedup, eliminate the memory limitations posed by the simulation on a uniprocessor and minimize the overhead inherent in a distributed simulation. Simulation speedup in a distributed Verilog simulator is possible only if the inherent parallelism in the simulated circuit can be extracted while the synchronization overhead is kept minimal. Without our optimistic techniques applied to the distributed Verilog simulation, the distributed Verilog simulation could be slower than the sequential Verilog simulation, as described in the preliminary experiment in chapter 4.

7.1 Thesis Contribution

The main contributions of this thesis are the following:

- Construction of an object-oriented framework for distributed Verilog simulation. This framework provides us with a vehicle to experiment with the synchronization and partitioning techniques which lie at the heart of distributed Verilog simulation

DVS has proved to be a good experiment environment in the distributed simulation lab. Besides being used for the experiments mentioned in this thesis, it has been utilized by other students[97] in distributed simulation lab for optimization of Time Warp with learning automata.

- Event reconstruction was developed as an optimization technique used to reduce memory consumption in Time Warp

  State saving and restoration and event saving add considerably to the synchronization cost. Due to the fine event granularity of Verilog simulation and and the extremely large number of events, the cost for event saving could be even larger than the state saving. If the memory consumption problem is not handled properly in distributed Verilog simulation, the simulation may not even finish.

- A design-driven iterative partitioning algorithm which takes advantage of the Verilog design hierarchy. The algorithm attains a compromise between minimal cutsize and load balance resulting in a good speedup.

  In our experiments, we found that excessive communication causes more rollbacks and delays GVT computation. Thus, the enormous message traffic consumes the memory so fast that it eventually kills the simulation. In order to minimize the message traffic between different partitions, the cutsize needs to be minimized while load balance between partitions is well maintained.

  A Verilog design has hierarchies which can be utilized to exploit circuit parallelism and minimize cutsize. However, existing partitioning algorithms for logic simulation usually work on a flattened netlist. A partitioning algorithm without

the design information as the guide lead to huge cutsize which makes the distributed Verilog meaningless because it is even slower than the sequential Verilog simulation.

7.2 Future Directions for Work

- Replicated logic to minimize cutsize

  In a real industrial level design, there are always some backbone components which are not easy to break into any partition. For example, the register block is supposed to provide setup value and parameters to all other blocks. The fanout from the register block to all other blocks is huge. Moreover, the communication from the register block to other blocks are enormous if the register read and write are frequent. Figure 7–1 shows such a circuit. Block0 is linked to all of the other blocks, Block1, Block2, Block3 and Block4.

  Fortunately the logic in the register block is normally small. Therefore, we could be able to replicate it in both partition 1 and partition 2, as shown in figure 7–1. In this way, the communication will be reduced significantly. This kind of technique is already employed in FPGA partitioning. However, to date it has not been used in the distributed Verilog simulation.

  A difficulty in implementing replicated logic lies in identifying it. The block which can be replicated needs to be small in terms of the gate count but its interconnection with other blocks is significant. Currently the replicated logic in an FPGA partition is done by the engineers manually. It will be interesting to automate the partitioning with the replicated logic in distributed Verilog simulation.

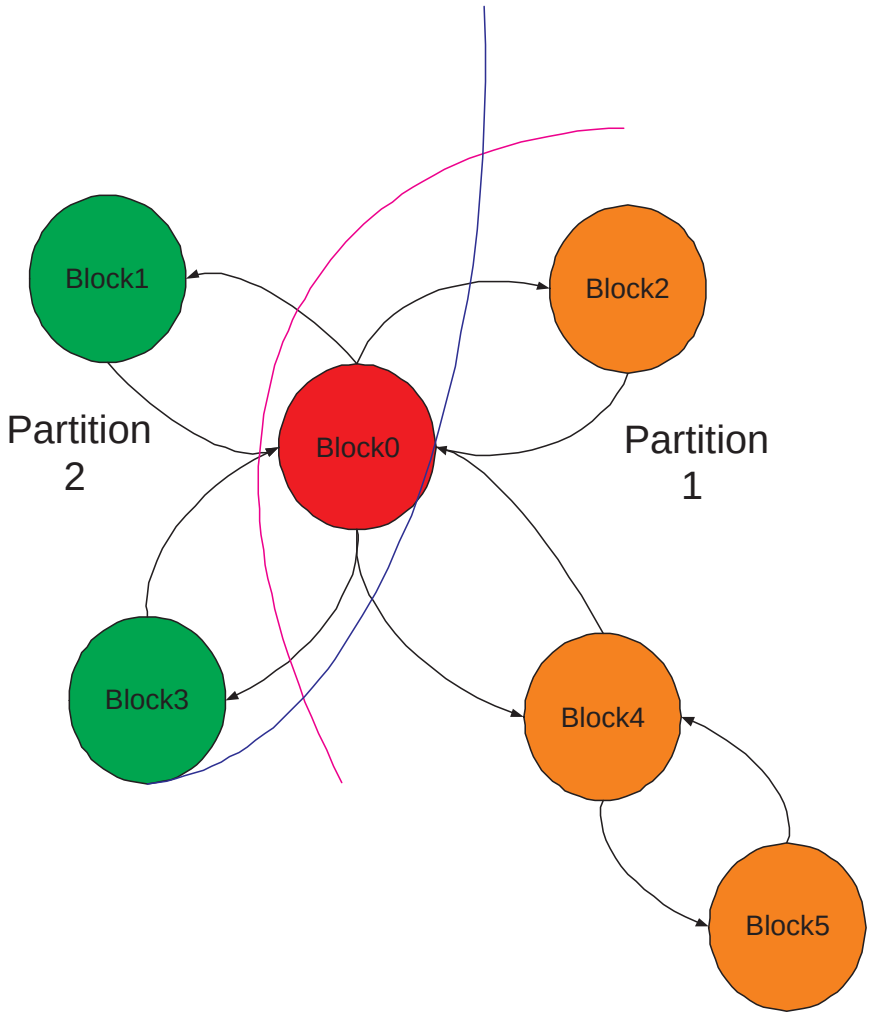


Figure 7–1: Replicated logic in the partitioning

- Rollback reduction techniques

  Smart rollback filter could be applied to reduce rollback. The basic idea of the smart rollback filter is to take advantage of the properties of the logic gates. For example, if one input of an AND gate is 0, the output of the AND gate will be 0 no matter what the values of other input are. If the rollback happens in one of the

input ports of the AND gates and the value of the other input port remains 0, the rollback will be ignored since it will not affect the final output of the AND gate.

- Faster GVT computation algorithm

  In the experiment, we noticed excessive memory consumption for partitions generated by DFS, BFS and random partitioning algorithm. During the simulation, the memory consumption keeps growing until the system runs out of memory and the simulation process crashes.

  After debugging and analyzing the simulator behavior, we found that the excessive memory consumption is caused by slow GVT computation, which uses Mattern's GVT algorithm[36] to computer GVT in OOCTW. The performance degrading of this algorithm is caused by too many messages between the simulation processes. After the first cut is initialized in the Mattern's GVT computation, all the machines need to collect all white messages to construct the second consistent cut. However, for the simulation with partition generated by DFS, BFS and random partitioning, too many white messages need to be collected, thus making the GVT computation very slow. Even worse, the simulation is still running at the full speed, generating an enormous number of new events, which will be inserted into the event queue. Since fossil collection is not executed efficiently because of the slow GVT computation, the memory consumed by the event queue and history state queue will eventually consume all memory in the system and kills the simulation.

  In the future, we need to put some research effort on the fast GVT computation algorithm so the efficient fossil collection could be executed in order to attack the memory consumption problem.

- Heuristic pre-simulation algorithm

  Currently we use brute force pre-simulation which runs all combinations of parameters k and b. In order to reduce the time devoted to pre-simulation, we are trying to develop heuristic pre-simulation algorithm which only runs a limited set of the k and b combinations. Basically the algorithm uses binary search or other heuristic search algorithm to speedup the search of the best balance point between communication and load balance.

  The disadvantage of the heuristic algorithm is that it could be trapped in the local minimum so it is not able to locate the best balance point. We will study further how to reduce the presimulation time without sacrificing the partitioning quality.

- Port DVS to shared memory machine or computer with dual/quad core

  As we know, the message passing in the distributed environment is the bottleneck for the distributed Verilog simulation. This is also the motivation for us to try to find out a better partitioning algorithm. However, on the other side, we could also take advantage of the shared memory machine in order to reduce the overhead of message passing. Moreover, as the machines with dual/quad cores become more and more popular and cost attractive, they could become a new research platform with the simulation processes running as threads in the computers.

REFERENCES

[1] Lijun Li, Hai Huang, and Carl Tropper. Dvs: an object-oriented framework for distributed verilog simulation. In *Parallel and Distributed Simulation, 2003. (PADS 2003)*, pages 173–180, June 2003.

[2] Lijun Li, Hai Huang, and Carl Tropper. Towards distributed verilog simulation. *International Journal of Simulation, Systems, Science & Technology*, 4(3-4):44–54, September 2003.

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

[4] Wenyong Deng Shantanu Dutt. Cluster-aware iterative improvement techniques for partitioning large vlsi circuits. *ACM Transactions on Design Automation of Electronic Systems(TODAES)*, 7(1):91–121, Jan 2002.

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

[6] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Transactions on VLSI Systems*, 7(1):69–79, 1999.

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

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

[9] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language Fourth Edition*. KLUWER Academic Publisher, 1992.

[10] Michael John Sebastian Smith, editor. *Application-Specific Integrated Circuits (The VLSI Systems Series)*. Addison-Wesley Professional, 1997.

[11] IEEE. *1076.1-1999 IEEE Standard VHDL Analog and Mixed-Signal Extensions*. 1999.

[12] Synopsys Inc. *Design Compiler*. http://www.synopsys.com/products/logic/design_compiler.html.

[13] Manuel A. d'Abreu. Gate-level simulation. *IEEE Design & Test*, 2(6):63–71, December 1985.

[14] the free encyclopedia WIKIPEDIA. *Hardware description language*. http://en.wikipedia.org/wiki/Hardware_description_language.

[15] Prithviraj Banerjee. *Parallel Algorithms for VLSI Computer Aided Design*. Prentice Hall, Inc., 1994.

[16] Michael Pidd. An introduction to computer simulation. In *Proceedings of the 26th conferences on winter simulation*, December 1994.

[17] Bernard P. Zeigler, editor. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.

[18] Franois E. Cellier, editor. *Continuous System Modeling*. Springer-Verlag, New York, 1991.

[19] *5Spice analysis software*. http://www.5spice.com.

[20] A. E. Ruehli, editor. *Circuit Analysis, Simulation and Design*. Elsevier Science Publishing Company, Inc., Amsterdam, North-Holland, 1986.

[21] IEEE, editor. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. The Institute of Electrical and Electronics Engineers, Inc., 1999.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[23] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computattions. *Communications of the ACM*, 24(11):198–206, November 1981.

[24] R. M. Fujimoto. Performance measurement of distributed simulation strategies. *Transactions of the Society for Computer Simulation*, 6:89–132, 1989.

[25] J. Misra. Distributed discrete event simulation. *ACM Computing Survey*, 18(1):39–65, November 1986.

[26] W. Cai and S. J. Tuner. An algorithm for distributed discrete-event simulation- the carrier null message approach. In *Proc. 1990 SCS Multiconference on Distributed Simulation*, pages 3–8, San Diego, California, 1990.

[27] A. Boukerch and C Tropper, editors. *Parallel Simulation on the Hypercube Multiprocessor*, volume 8. Springer-Verlag.

[28] Bojan Groselj. Fault-tolerant distributed simulation. In *Proceedings of the 23rd conference on Winter simulation*, 1991.

[29] David M. Nicol and Paul F. Reynolds Jr. Problem oriented protocol design. In *Winter Simulation Conference*, pages 471–474, 1984.

[30] Seng Chuan Tay, Yong Meng Teo, and Rassul Ayani. Performance analysis of time warp simulation with cascadingrollbacks. In *Parallel and Distributed Simulation, 1998. (PADS 1998)*, pages 30–37, May 1998.

[31] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[32] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Proc. Of the SCS Multiconference on Distributed Simulation*, pages 61–67, February 1998.

[33] H. Bauer and Sporrer C. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. In *Proc. of the 26th Annual Simulation Symposium*, pages 12–20. Socity for Computer Simulation, April 1993.

[34] D. West and K. Panesar. Automatic incremental state saving. In *Parallel and Distributed Simulation, 1996. (PADS 1996)*, pages 78–85, May 1996.

[35] M.Q.Xu S.J.Turner. Performance evaluation of the bounded time warp algorithm. In *Parallel and Distributed Simulation, 1992. (PADS 1992)*, pages 117–126, 1992.

[36] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[37] B. Samadi. *Distributed Simulation algorithms and performance analysis*. PhD thesis, University of California, Los Angelos, 1985.

[38] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism, part ii: Global control. Technical report, Rand Corporation, 1983.

[39] S. Bellnot. Global virtual time algorithms. In *Proceedings of the Multiconference on Distributed Simulation*, pages 122–127, 1990.

[40] A. I. Conception and S. G. Kelly. Computing global virtual time using the multi-level token passing algorithm. In *5th Workshop on parallel and distributed simulation(PADS'91)*, pages 63–68, 1991.

[41] B. R. Preiss. The yaddes distributed discrete event simulation specification language and environment. In *Proc. Of the SCS Multiconf. On Distributed Simulation*, pages 139–144, 1989.

[42] P.A. Wilsey J. Fleischmann. Comparative analysis of periodic state saving techniques in time warp simulators. In *Ninth Workshop on Parallel and Distributed Simulation (PADS'95)*, pages 50–58, June 1995.

[43] Christopher D. Carothers, Kalyan S. Perumalla, and Richard Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Workshop on Parallel and Distributed Simulation*, pages 126–135, 1999.

[44] P. Wilsey and A. Palaniswamy. Rollback relaxation: A technique for reducing rollback costs in an optimistically synchronized simulation. In *International Conference on Simulation and Hardware Description Languages, Society for Computer Simulation*, pages 143–148, January 1994.

[45] Y.-B. Lin and B. R. Preiss. Optional memory management for time warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283–307, 1991.

[46] Jr. J. Briner. *Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time*. PhD thesis, Duke University, 1990.

[47] Y. Matsumoto and K. Taki. Parallel logic simulation on a distributed memory machine. In *Proc. European Conference on Design Automation*, pages 76–80, 1992.

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

[49] 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.

[50] H. Bauer, C. Sporrer, and T.H. Krodel. On distributed simulation using timewarp. Technical report, Technical University of Munich, 1992.

[51] T. Krodel and K. Antreick. An accurate model for ambiguity delay simulation. In *Proc. EDAC*, pages 122–127, 1990.

[52] P. Luksch. Evaluation of three approaches to parallel logic simulation on a distributed memory multiprocessors. In *Proc. 26th Annual Simulation Symposium, Arlington*, pages 2–11, 1993.

[53] P. Luksch and H Weitlich. Timewarp parallel logic simulation on a distributed memory multiprocessor. In *Proc. SCS European Simulation Conference, Lyon*, pages 585–589, 1993.

[54] 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, 1995.

[55] R. L. Bagrodia and W.-T. Liao. Maisie: A language for the design of efficient discrete-event simulations. 20:225–238, 1994.

[56] L. Zhu et.al. Parallel logic simulation of million-gate vlsi circuits. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, 2005.

[57] Herve Avril and Carl Tropper. Scalable clustered time warp and logic simulation. *VLSI design*, 00:1–23, 1998.

[58] Stephen Williams. *Icarus Verilog*. http://icarus.com/eda/verilog.

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

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

[61] Dragos Lungeanu. *Discovery: Distributed simulation of digital and analog VLSI systems*. PhD thesis, University of Iowa, July 2000.

[62] L.W. Nagel. *SPICE2: A computer program to simulate semiconductor circuits*. PhD thesis, U.C. Berkeley, Electronics Research Laboratory Rep. No. ERL-M520, May 1975.

[63] Qing Xu and Carl Tropper. Xtw, a parallel and distributed logic simulator. *Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 1064–1069, 2005.

[64] P. A. Wilsey, D. E. Martin, and K. Subramani. Savant/tyvis/warped: Components for the analysis and simulation of vhdl. In *VHDL Users' Group Spring 1998 Conference*, pages 195–201, 1998.

[65] S. Subramaninian et al. Study of a multilevel approach to partitioning for parallel logic simulation. In *IPDS00*, May 2000.

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

[67] S. Levitan. *Vcomp and Vsim Reference Manual*. University of Pittsburgh.

[68] Tun Li, Yang Guo, and Si-Kun Li. Design and implementation of a parallel verilog simulator: Pvsim. In *Proceedings of the 17th International Conference on VLSI Design (VLSID'04)*, pages 173–180, 2004.

[69] M. L. Briner Bailey and Chamberlain. Parallel logic simulation of vlsi systems. In *ACM Computing Surveys*, volume 26, pages 255–294, Sept 1994.

[70] Lijun Li and Carl Tropper. Event reconstruction in timewarp. In *Parallel and Distributed Simulation(PADS)*, pages 37–44, 2004.

[71] Yi-Bing Lin, Bruno R. Preiss, Wayne M. Loucks, and Edward D. Lazowska. Selecting the checkpoint interval in time warp parallel simulation. In *Proc. 1993 Workshop on Parallel and Distributed Simulation*, pages 3–10. Institute of Electrical and Electronics Engineers, May 1993.

[72] S. Bellenot. State skipping performance with the time warp operating system. In *6th Workshop on Parallel and Distributed Simulation*, pages 53–61. Socity for Computer Simulation, January 1992.

[73] Hai Huang. A partitioning framework for distributed verilog simulation. Master's thesis, School of Computer Science, McGill University, 2003.

[74] A. B. Kahng A. E. Caldwell and I. L. Markov. Design and implementation of the fiduccia-mattheyses heuristic for vlsi netlist partitioning. In *Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX), Baltimore*, pages 177–193, Jan. 1999.

[75] R.Chamberlain and C.Henderson. Evaluating the use of pre-simulation in vlsi circuit partitioning. In *PADS94*, pages 139–146, 1994.

[76] Swaminathan Subramanian, Dhananjai M. Rao, and Philip A. Wilsey. Applying multilevel partitioning to parallel logic simulation. In *Parallel and Distributed Computing Practices*, volume 4, pages 37–59, March 2001.

[77] Vipin Kumar George Karypis, Rajat Aggarwal and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. In *ACM/IEEE Design Automation Conference*, pages 526–529, 1997.

[78] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

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

[80] Naraig Manjikian and Wayne M. Loucks. High performance parallel logic simulations on a network of workstations. In *Proceedings of the seventh workshop on Parallel and distributed simulation*, pages 76–84, May 1993.

[81] Chamberlain R. D. and Henderson C. Evaluating the use of presimulation in vlsi circuit partitioning. In *Proc. 1994 Workshop on Parallel and Distributed Simulation*, pages 139–146, 1994.

[82] L. Soule. *Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms*. PhD thesis, Standford University, 1992.

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

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

[85] B. W. Kernighan and S. Lin. An efficient heiristic procedure for partitioning graphs. *Bell System Tech. Journal*, 49:291–307, 1970.

[86] D. G. Schweikert and B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th Design Automation Workshop*, pages 57–62, 1972.

[87] B. Krishnamurthy. An improved min-cut algorithm for partitioning vlsi networks. *IEEE Trans. on Computers*, 33(5):438–446, 1984.

[88] L. A. Sanchis. Multiple-way network partitioning. *IEEE Trans. on Computers*, 38(1):61–81, 1989.

[89] C.I. Park and Y.B. Park. An efficient algorithm for vlsi network partitioning problem using a cost function with balancing factor. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 12(11):1686–1694, 1993.

[90] A. Dasdan and C. Aykanat. Two novel multiway partitioning algorithms using relaxed locking. *IEEE Trans. on Computer-Aided Design*, 16(2):169–178, 1997.

[91] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. 37(6):865–892, 1989.

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

[93] H. H. Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1539, 1996.

[94] Chau-Shen Chen, Ting Ting Hwang, and C. L. Liu. Architecture driven circuit partitioning. In *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, volume 9, pages 383–389, April 2001.

[95] K.-H. Chang, H.-W. Wang, Y.-J. Yeh, and S.-Y. Kuo. Automatic partitioner for distributed parallel logic simulation. In *Modelling, Simulation, and Optimization*, volume 429, Aug 2004.

[96] Jong-Sheng Cherng, Sao-Jie Chen, Chia-Chun Tsai, and Jan-Ming Ho. An efficient two-level partitioning algorithm for vlsi circuits. In *Asia and South Pacific Design Automation Conference 1999 (ASP-DAC'99)*, pages 69–72, 1999.

[97] Jun Wang and Carl Tropper. Optimizing the time warp protocol with learning automata. *European Simulation and Modelling Conference*, 2007.