May 1989

UILU-ENG-89-2214
CSG-103

COORDINATED SCIENCE LABORATORY
College of Engineering

A PARALLEL ALGORITHM FOR SWITCH-LEVEL TIMING SIMULATION ON A HYPERCUBE MULTIPROCESSOR

Hariprasad Nannapaneni Rao

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-89-2214 (CSG-103)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center Hampton, VA 23665
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG 1-613
8c. ADDRESS (City, State, and ZIP Code) See 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Parallel Algorithm for Switch-Level Timing Simulation on a Hypercube Multiprocessor.-

12. PERSONAL AUTHOR(S)
Hariprasad Nannapaneni Rao

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1989, April.-	29

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Parallel algorithms, timing simulation, virtual time, rollback, event-driven simulation and partitioning.-

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis is concerned with the parallel approach to speeding up simulation, specifically the simulation of digital LSI MOS circuitry on the Intel *i*PSC/2 hypercube. The simulation algorithm is based on RSIM, an event-driven switch-level simulator that incorporates a linear transistor model for simulating digital MOS circuits. Parallel processing techniques based on the concepts of Virtual Time and rollback are utilized so that portions of the circuit may be simulated on separate processors, in parallel for as large an increase in speed as possible. A partitioning algorithm is also developed inorder to subdivide the circuit for parallel processing.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

A PARALLEL ALGORITHM FOR SWITCH-LEVEL TIMING SIMULATION ON A HYPERCUBE MULTIPROCESSOR

BY

HARIPRASAD NANNAPANENI RAO

B.S., University of Illinois, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1989

Urbana, Illinois

ABSTRACT

This thesis is concerned with the parallel approach to speeding up simulation, specifically the simulation of digital LSI MOS circuitry on the Intel *i*PSC/2 hypercube. The simulation algorithm is based on RSIM, an event-driven switch-level simulator that incorporates a linear transistor model for simulating digital MOS circuits. Parallel processing techniques based on the concepts of Virtual Time and rollback are utilized so that portions of the circuit may be simulated on separate processors, in parallel for as large an increase in speed as possible. A partitioning algorithm is also developed inorder to subdivide the circuit for parallel processing.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. Prithviraj Banerjee, for his immense patience and wisdom in seeing me through to the completion of this thesis. In addition, I would like to thank Tom Fruchterman for his help in making the 4000 and some odd lines of code work. Finally, I would like to give my love and thanks to my parents and entire family for their love, support, and patience over these twenty-three years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Due to the expense and complexity associated with fabricating a VLSI device, simulation is used extensively in the design of VLSI circuits. A device can be simulated after the designer has entered a complete description of the circuit into a computer. A circuit description consists of a set of components, such as transistors in the case of MOS devices, and a set of nodes with no restrictions on interconnections. The nodes are considered as both the terminals of the components and the wires which interconnect the components. Current is then assumed to flow from component to component through common nodes.

Simulation is becoming more and more popular, and new applications for simulators are constantly being developed. However, the technology of semiconductor device fabrication has been improving such that it is now possible to fabricate MOS devices that contain more than 1,000,000 transistors. Chips of these sizes cannot be adequately tested with current simulator technology in a reasonable amount of time. Consequently, many new ideas are being implemented in the area of circuit simulation in hopes of meeting the increased simulation requirements imposed by the larger and more powerful VLSI circuits.

Several approaches to speeding up circuit simulation have been developed to meet the challenge of simulating large VLSI circuits. The use of special purpose hardware built to run simulations, such as the Yorktown Simulation Engine, developed by IBM [1], and Abramovici's simulation pipeline [2], is one approach to this problem. However, these types of machines tend to be very expensive and limited to a narrow class of problems. Other approaches have led to the improvement of existing simulation algorithms or to the development of new ones that run on expensive supercomputers such as the Cray. A relatively recent approach, however, has been the usage of general purpose parallel computers to run existing simulation algorithms [3].

1.2 Overview

This thesis is concerned with the parallel approach to speeding up simulation, specifically the simulation of digital LSI MOS circuitry on the Intel *i*PSC/2 hypercube. The simulation algorithm is based on RSIM, an event-driven switch-level simulator that incorporates a linear transistor model for simulating digital MOS circuits [4]. Parallel processing techniques based on the concepts of Virtual Time and rollback [5] are utilized so that portions of the circuit may be simulated on separate processors, in parallel, for as large an increase in speed as possible. A partitioning algorithm is also developed in order to subdivide the circuit for parallel processing.

The following chapter is a brief overview of simulation and the RSIM simulation algorithm. Chapter 3 discusses parallel strategies and the notions of Time Warp for the speedup of the simulation process. A circuit partitioning algorithm based upon the calculations that RSIM performs is developed in Chapter 4. Chapter 5 contains a description of the hypercube and of the parallel simulation that was developed for it. Results of the experiments performed on the parallel simulation are given along with concluding remarks in Chapter 6.

CHAPTER 2

SWITCH-LEVEL TIMING SIMULATION

2.1 Simulation at Different Levels of Detail

The two fundamental approaches to circuit simulation historically have been either analytical or functional. The former involves detailed circuit analysis and is frequently referred to as SPICE-level [6] simulation whereas the latter is known as logic-level simulation.

The majority of SPICE-level simulators calculate transient circuit response by numerically integrating a complete set of simultaneous nonlinear differential equations. Using these types of circuit analyzers, in which circuit simulation is based on physical component models, the behavior of a circuit at sub-nanosecond detail can be readily observed. Information such as the internal currents flowing through discrete components and the amount of heat dissipated through discrete components can also be obtained. Most physical considerations are taken into account, allowing the designer a high level of confidence that his circuit will work once it is fabricated.

Unfortunately, there is a high computational cost in providing this level of detail. SPICE-level simulation times are proportional to n^m, where n is the number of nonlinear devices in the circuit, and m is between one and two [7]. Newer simulators such as SPLICE [8] and RELAX [9] offer the same level of detail in less time by exploiting the unidirectional nature of MOS devices and the spacial sparsity and temporal sparsity of SPICE-level matrix calculations. Still, these types of simulators show only one to two orders of magnitude speedup over SPICE-level simulators and as such are only adequate for the simulation of a circuit with less than 10,000 devices [10].

Functional simulators use a less detailed approach: primitive models such as NOT, AND, OR, NAND, and NOR gates are used rather than analog component models based on physical characteristics. Logic-level simulators generally calculate logic levels only, although timing parameters may be assigned to the logic models based on known circuit parasitics. A problem does arise, however, when attempting to

model the behavior of MOS circuitry; gate models do not contains the same bidirectional switching elements, which permit currents to flow through them in either direction, that characterize MOS devices. This quality cannot be directly simulated using simple combinations of logic gates. Therefore, gate-level simulators are not sufficiently accurate in simulating the discrete features of MOS circuitry.

Switch-level simulators such as LAMP [11] and MOSSIM [12] combine very simple models of transistors with efficient event based simulation algorithms to directly simulate MOS circuits. These types of simulators are very fast because of the simplicity of the models used and are therefore excellent at simulating the switching behavior of large MOS circuits. Unfortunately, also because of the simplicity of the models used, these methods are not well-suited for detailed circuit analysis as no timing or voltage level information is calculated. It should be noted that not all circuits can be simulated with this type of a simulator, *e.g.*, the transmission gate XOR. Therefore, results from switch-level simulators should be used only for predicting the gross behavior of the device, and not whether the device will work properly once it is etched in silicon.

RSIM [13], developed at MIT, provides a middle ground between SPICE-level simulators and switch-level simulators. RSIM offers comparative timing information at speeds much faster than circuit analyzers through a hybridization of switch-level simulation and circuit analysis. This allows RSIM to simulate significantly larger circuits at a level of detail sufficient to determine which sections require further and more detailed analysis.

2.2 RSIM - A Logic-Level Timing Simulator

RSIM is a logic-level simulator that uses a simple linear model: the result falls between switch-level and timing simulations. RSIM adopts a linear model of the transistor in terms of an effective resistance, R_{eff}, comprised of a variety of parameters, including the average channel resistance of the transistor over its range of operating terminal voltages and the context of the transistor's use. The transistors are realized with a simple bidirectional linear model which may be viewed as a resistor and switch in series and can be described as

$$R_{ds} = \begin{cases} R_{eff} & \text{(switch closed)} \\ \infty & \text{(switch open)} \\ [R_{eff}, \infty] & \text{(switch in unknown state)} \end{cases}$$

In RSIM, the problem of determining the value of a node is handled by first deriving the *Thevenin* equivalent circuit (see Figure 2.1) and and then determining the value of V_{thev} in relation to the quantum levels denoting 0, 1, and X. If this value differs from the node's previous value, a transition for the node's value to change is scheduled $R_{drive}C_{load}$ time steps in the future. This provides the designer with both a functional analysis of the circuit and some relative timing information.

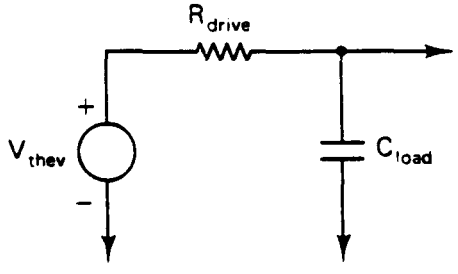


Figure 2.1. Thevenin Equivalent Circuit.

2.3 The RSIM Simulation Algorithm

The main simulation algorithm is *event* based. An event specifies (i) a node in the network, (ii) a new logic state, and (iii) a time at which the node's value is to be changed to the new logic state. The *eventlist* is a time-sorted list of events. Processing an event entails

(a) removing the event from the eventlist

(b) changing the node's state to reflect its new value

(c) calculating any consequences, *i.e.*, new events resulting from the node's new value. First, all nodes that could be affected by the change are found and marked - this requires a tree-walk of the network starting at the source and drain nodes of transistors for which the changing node is the gate. The

tree-walk follows sources/drain connections, stopping at input nodes or non-conducting transistors. For each marked node two calculations are made: (1) a charge-sharing calculation that models changes of state due to charging/ discharging of node capacitance and (2) a final-value calculation that determines the node's ultimate state.

As seen in step (c) the network is naturally partitioned into stages, each stage consisting of nodes shorted together by source-drain connections. This allows RSIM to provide a good base for the exploitation of parallelism in digital MOS circuit analysis.

CHAPTER 3

PARALLEL SIMULATION

3.1 Overview

Several issues are germane to the study of parallel or distributed algorithms. The first issue is *task allocation*, that is, the breakdown of the total workload into smaller tasks assigned to different processors. This topic will be covered in Chapter 4 on partitioning of circuits. Another issue is the *communication* of the interim computational results between processors. Coverage of this topic is deferred until the hypercube is introduced. A third issue is the *synchronization* of the computations of different processors. Certain methods require processors to wait at predetermined points, or *phases*, for the completion of computations or for the arrival of data. Methods such as these are often classified as *synchronous*. Other methods, namely *asynchronous*, have no waiting period; therefore, the validity of the results from these methods must be assessed.

A synchronous algorithm is mathematically equivalent to an algorithm governed by a global clock, *i.e.,* an algorithm for which the start of each phase is simultaneous for all processors, and the end of the message reception is simultaneous for all messages. In attempting to implement a synchronous algorithm in an asynchronous distributed system, a *synchronization mechanism* or a *synchronizer* is required. There are three fundamental approaches to resolving this problem. *Global synchronization*, as previously described, is used in synchronous algorithms, or *lock-step* simulations. *Local synchronization* is used in the Chandy-Misra method [14] and many other distributed simulation algorithms [15] [16]. The third approach, *rollback*, has been popularized for its use in the Time Warp mechanism [17].

3.2 Lock-step Simulation

The lock-step approach is usually implemented using a global counter that is initially set to the number of processors running the simulation at time step i and decremented each time a processor

completes time step *i*. When the counter reaches zero, all processors then communicate shared data among themselves, essentially resynchronizing themselves with one another. All processors then proceed en masse to simulate the next time step, $i+1$.

Several problems exist with this method. Most importantly, the simulation will proceed only at the rate governed by the slowest processor for each time step. Equivalently, all but one processor will be idle at some point during the simulation time step. Therefore, the work among the processors must be *load balanced*, or divided as evenly as possible. It will be shown in the next chapter that this problem cannot be easily solved; therefore, methods that rely heavily on load balancing may deviate considerably from their ideal behavior.

An interesting effect of improper load balancing for this method is that the addition of an extra processor to the simulation does not necessarily increase the performance of the simulation. If the additional processor is not assigned a portion of workload that is creating the bottleneck, the processor will eventually join the pool of idle processors waiting for the bottleneck to resolve. In this case, which may occur quite often, no additional increase in parallelism will occur for the simulation, and therefore, the system is no longer scalable.

3.3 Chandy-Misra Method

Chandy and Misra's approach to distributed simulation is to allow phases of a problem to be executed in parallel, *i.e., a sequence of parallel computations*, and then to require sychronization at phase interfaces. Since there is no centralized process that oversees the network, the termination of a phase, namely *deadlock*, is detected in a distributed manner. The network proceeds until deadlock occurs, then detects deadlock, and finally will recover from deadlock and resume computations.

Physical processes (PPs) that interact with one another are modeled as logical processes (LPs) that communicate with one another via messages. A message m from PP_i to PP_j at time t is simulated by LP_i sending LP_j a tuple: (t, m). The effect of encoding time in

the message results in synchronization without a global clock. This method also requires that a message sent by LP_i at time t is a function of its initial state, t, and the messages it has received up to and including t. This is referred to as the *realizability condition*.

The Chandy-Misra method can be viewed as a network consisting of logical processes with *directed channels* or *arcs* connecting pairs of processes. Interactions between LPs in the network consist of time stamped messages moving along the network's arcs. Each LP executes sequential code and two special commands, receive and send, until deadlock occurs. Deadlock occurs either when any LP_i is waiting to receive a message from some LP_j and there is no message in transit from LP_j to LP_i or when every LP in the network is terminated. Chandy and Misra postulate a *null* message: (*t*, *null*) sent by LP_j to LP_i to announce the absence of messages. LP_i is then defined to be *resumable* if the set of lines it waits for at this point is different from the set of lines it was waiting for at the point of deadlock.

Several problems may exist with this model. Because *blocking* of LPs occurs while sending or receiving messages, LPs contained within an undirected cycle in the network may be innundated with null messages. In addition, deadlock around a cycle can rapidly become global. Seethalakshmi [18] has shown that the Chandy-Misra method and other similar distributed algorithms will approach ideal performance when there are no multiple loops in the network. This is not the case, however, in VLSI circuit design. Therefore, this method is considered too conservative to extract concurrency for the simulation of VLSI circuits.

3.4 Time Warp Model

The Time Warp mechanism is an optimistic, loosely-coupled, distributed, discrete-event synchronization algorithm. Time Warp allows processes to execute independently according to *local virtual clocks* without requiring each process to wait on the local virtual clock of other processes that communicate with it. Conflicts in time which result from allowing local virtual clocks of each process to become unsynchronized are resolved by the use of the rollback mechanism. Each processor will

continue computing at its own pace under the assumption that no message is received. In such cases of conflict, the rollback mechanism allows the processor to invalidate its computation by restoring an old *checkpoint*, and restart itself, taking the received message properly into account. Messages that cause conflicts are called *stragglers*, and messages sent by the processor during erroneous computation are cancelled by *antimessages*.

The Time Warp mechanism is characterized in [19] by Berry as follows:

3.4.1 Data structure of a time warp object

A Time Warp object is represented as five main data structures:

1. a state
2. an input message queue
3. an output message queue
4. a past state queue
5. a local virtual time clock

The input message queue contains all of the messages that have been received by the object since the last *garbage collection*, including messages that have already been processed. The reason for keeping the processed message is that when a rollback occurs, the messages must be reprocessed.

The output message queue of an object contains all of the messages the object has sent since the last garbage collection. If the object rolls back its computation, all the messages it sent to other objects from the simulation time to which it rolls back until the present can be found in the output queue so that proper antimessages can be generated.

The past state queue is an ordered list of checkpoint states. When a straggler arrives, the most recent past state which is earlier than the straggler's timestamp is the state to which the object is rolled back.

The local virtual time is the object's local simulation clock. During the execution of an event, it represents the simulation time at which the event occurs. Otherwise, it contains the lowest timestamp

of all the unprocessed messages. If there is no unprocessed message in the input queue, the local virtual time is set to infinity. When a message is inserted into the object's input queue, its timestamp is compared with the local virtual time. If the former is less or equal, a rollback is initiated. The simulation is terminated when all the local virtual times are infinity.

3.4.2 Antimessages

A negative copy of every message sent by an object is kept in the object's output queue. A Time Warp message has a field that denotes whether it is a positive, regular message or a negative, antimessage.

The sending of an antimessage can be regarded as releasing the antimessage from the output queue and transmitting it to the destination of the original message. When the antimessage is received it annihilates with the original message. In addition, if the antimessage is timestamped earlier than the receiving object's local virtual time, then a rollback is initiated.

3.4.3 Rollback

A rollback is an action taken in response to the arrival of a straggler at an object. The object retrieves the most recent state with a local virtual time lower than the straggler's timestamp. The local virtual time of the object is set equal to the local virtual time of the retrieved state. The restored local virtual time determines which of the messages in the input queue must be reprocessed and which of the antimessages in the output queue must be released. There are two ways of releasing antimessages, either by *aggressive cancellation* or by *lazy cancellation*.

3.4.4 Cancellation methods

Aggressive cancellation means that the antimessages for all messages sent during a rolled-back simulation time period are released as soon as the object is rolled back. The advantage of this method is that when messages are wrong the object reports them wrong as soon as it can, thereby canceling their

effects as soon as possible. Rapid cancellation can prevent cascading of antimessages and, hence, rollbacks, because incorrect messages are prevented from creating other incorrect messages. The disadvantage of aggressive cancellation is that the object may retransmit the same messages that it canceled. In this case other objects may be rolled back twice, unnecessarily.

In lazy cancellation, the object waits until the send time of the output message has been reached again after rollback before deciding whether to cancel it. At that time, the antimessage is released only if the message that was sent before does not match the newly created message.

Jefferson [20] and Gafni [21] have shown that lazy cancellation has a number of advantages over aggressive cancellation. The authors suggest that for most applications, lazy cancellation will produce fewer antimessages and secondary rollbacks than aggressive cancellation. For these cases, lazy cancellation will offer substantially better performance. Also in certain situations, aggressive cancellation can lead to infinite rollback and the mechanism will subsequently deadlock.

3.4.5 Global virtual time

The global virtual time is the minimum of the local virtual times of all objects and of the timestamps of all messages in transit. In order to obtain an accurate global virtual time, it would be necessary to stop the computation until all local virtual times and relevant timestamps are collected. However, stopping the computation would defeat the purpose of Time Warp. Therefore, a pessimistic estimate obtainable without halting the computation is used as the global virtual time estimate that is distributed to each object.

The importance of the global virtual time is that no rollback can occur for a simulation time earlier than the global virtual time, since no message can be stamped with simulation time lower than its sender's local virtual time. Therefore, output produced before the global virtual time can be committed, and messages and states with stamps earlier than the global virtual time can be garbage-collected. In addition, when the global virtual time is equivalent to the termination time of the simulation, the simulation is completed.

3.4.6 Performance

Jefferson has performed a Pool Balls benchmark test using the Caltech/JPL Mark III Hypercube running under the Time Warp mechanism [22]. Results from the simulation indicate a speedup of 7 using 8 processors and 12 using 16 processors relative to the performance of the simulation using one processor. These results are quite encouraging and an attempt will be made to achieve similar performance for the parallel simulation of VLSI circuits using the concepts of Time Warp.

CHAPTER 4

CIRCUIT PARTITIONING

4.1 Overview

The need to partition MOS circuits for parallel simulation is relatively new; however, people have long been interested in the problem of partitioning networks or graphs. The ideas developed from partitioning of graphs can readily be applied to partitioning of circuits.

A graph, denoted G = {N,E}, consists of a finite set of nodes (or vertices) and a set of pairs of vertices, E, called edges. Nodes in an electrical circuit may be modeled as vertices in a graph and transistors as edges between the source and drain nodes. S is a *stage* or *charge-coupled partition* of N if and only if

(1) $N = S_1 \cup S_2 \cup S_3 \cup \cdots$

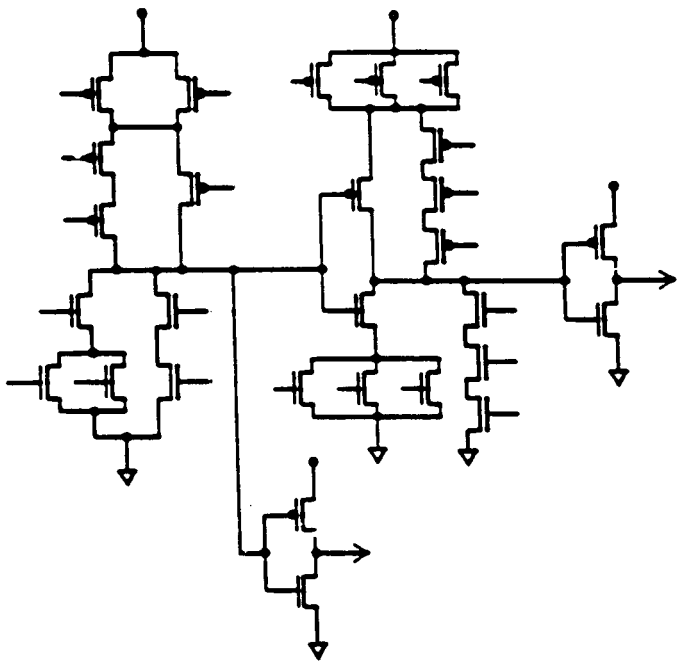
(2) $S_i \cap S_j = \phi$, for each i and j.

and is formed using the algorithm of Figure 4.1.

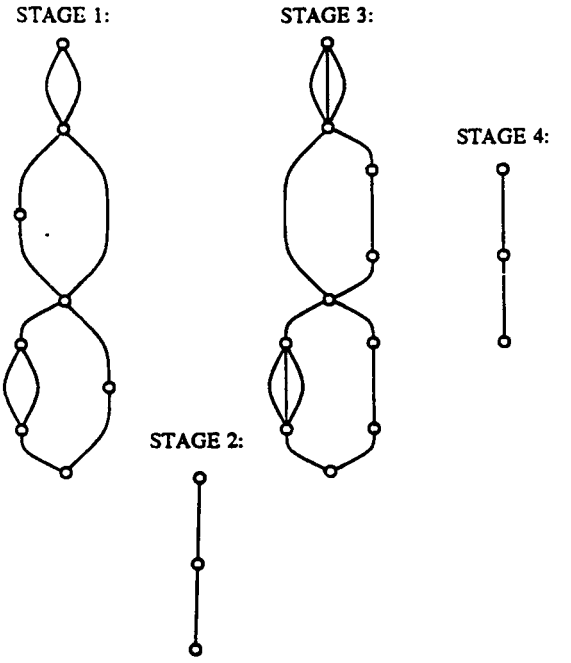
```
Calculate_stages {
    For each unprocessed node n in the circuit {
        Assign n to a new group
        For each transistor attached to n by its source or drain {
            Add unprocessed node, m, that is attached to drain or source of
                transistor, respectively, to group
            Recursively trace through sources and drains attached to m
        }
    }
}
```

Figure 4.1. Algorithm to Compute Stages.

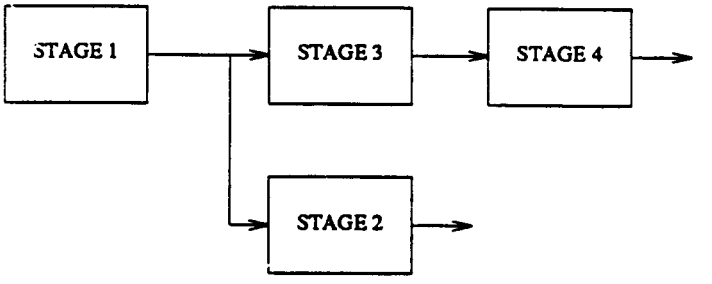
Figure 4.2 shows an example of the graph representation for an MOS circuit and the division of stages for the circuit. Since the calculations RSIM performs to compute a node's value always involve knowing the values of the other nodes in its stage, a stage is never broken apart and is the smallest piece of a circuit in which we have interest.



(a) Example Circuit



(b) Graph Representation of Circuit



(c) Block Diagram of Stages

Figure 4.2. Example Circuit Divided into Stages

A group, denoted P, is a collectively and mutually exclusive collection of stages. The problem, borrowed from graph theory, is to construct groups such that there is minimal interconnections between the groups and such that the stages are evenly distributed over the groups. These two requirements correspond to limiting interprocessor communication and balancing the load on the processors [23].

4.2 Different Approaches

Since the solution of this problem is known to be NP-complete, most approaches involve heuristics designed to come close to optimality, though not guaranteed to produce the optimal solution [24]. These heuristics tend to be very complex, and computation intensive. The pairwise optimization procedure by Kernighan and Lin [25] runs in $O(n^2 \log n)$ time.

On the other computational extreme, random partitioning will run in O(n log n) time. However, this method results in enormous interprocessor communication. Partitioning by fanin and fanout cones, methods commonly used in partitioning of VLSI circuits for parallel simulation, requires circuit leveling and loop detection and removal [26]. This constraint is not acceptable when attempting to simulate the two-phase clocking of MOS circuitry.

4.3 Modified Element Strings Approach

The element strings approach [27] consists of (1) dividing the circuit into a set of connected stages with at most one fanin and one fanout included and (2) balancing the loads across all of the partitions. The purpose of dividing the circuit up in this manner is to maximize the likelihood of concurrent simulation activity resulting from a signal change on the driving output. Although this scheme provides high concurrency levels, large amounts of interprocessor communication also occur.

A simple modification to the above algorithm maintains the goal of statically load balancing the partitions while limiting interprocessor communication. Grouping stages with the largest fanin and fanout with one another will substantially reduce the interprocessor communication. Externally generated events from a stage will have the highest probability of affecting stages that are contained within the

same group, as stages with the greatest dependencies on one another will in all likelihood be merged together. The algorithm for creating groups is given below in Figure 4.3. Figure 4.4 shows an example of groups constructed from stages.

```
Merge_stages {
    Calculate_stages
    For i = 1 to number of processors {
        While (size of group i < average transistor count)
        {
            Add stage with largest fanin and fanout connection to group i
        }
    }
}
```

Figure 4.3. Creation of Groups from Stages.

As the dynamic behavior of a circuit cannot be predicted without statistical observation of the behavior of a circuit, a simple approach to static load balancing was used. A stage cannot be added to a group if the *average transistor count* (total number of transistors in the circuit ÷ number of processors) has been exceeded. Although the algorithm will allow groups of size *2 * (average transistor count - 1)*, it is expected that most sizes of stages will be in the vicinity of 3 to 10 transistors; hence, the load for a given group should not fluctuate from the average transistor count by more than this amount.

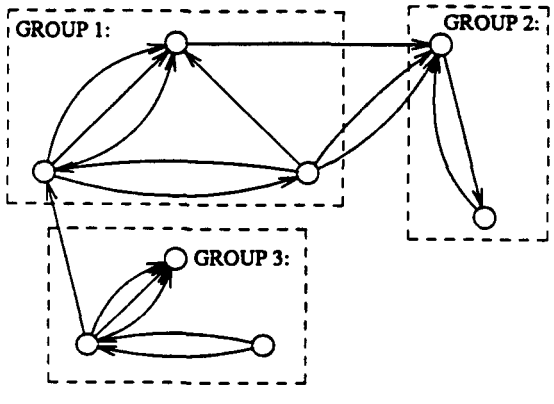


Figure 4.4. Example of Groups Constructed from Stages.

CHAPTER 5

HYPERCUBE IMPLEMENTATION

5.1 Overview of Hypercube System Architecture

The hypercube is an example of a distributed-memory, message-passing multiprocessor in which the processors have only private local memory and their activities are coordinated by sending messages among themselves through an N-cube interconnection topology. In an N-dimensional hypercube (also called the binary N-cube, cosmic cube, etc.), 2^N processors are consecutively numbered by binary integers from 0 through 2^N-1. Each processor is connected to all of the other processors whose binary tags differ from its own by exactly one bit. Topologically, this places each processor at the vertices of an N-dimensional cube (see Figure 5.1). An N-dimensional hypercube can be constructed recursively by taking two (N-1)-dimension hypercubes and connecting each node in one to the corresponding node in the other with a bidirectional link (see Figure 5.2). Each node has exactly N adjacent neighbors in this topology. The average distance between nodes is N/2 and the maximum distance is N.

An attractive feature of the hypercube is its homogeneity; the entire system looks topologically identical from the point of view of every node. In addition, the hardware complexity of the hypercube architecture grows Nlog(N) since there are only Nlog(N) links connecting the processors. This is in contrast to N^2 complexity for fully-connected architectures.

Although the hypercube is an MIMD (multiple instruction, multiple data) machine, one of the most popular programming styles for it has been characterized as *single program, multiple data*. In these applications, all of the processors run the same program, but they do not execute instructions in the lock-step manner of an SIMD (single instruction, multiple data) machine [28].

5.2 Hypercube Software Environment

The typical hypercube software application has a program that runs on the host and a program that runs on the nodes. The part of the application that is the host program executes as one or more

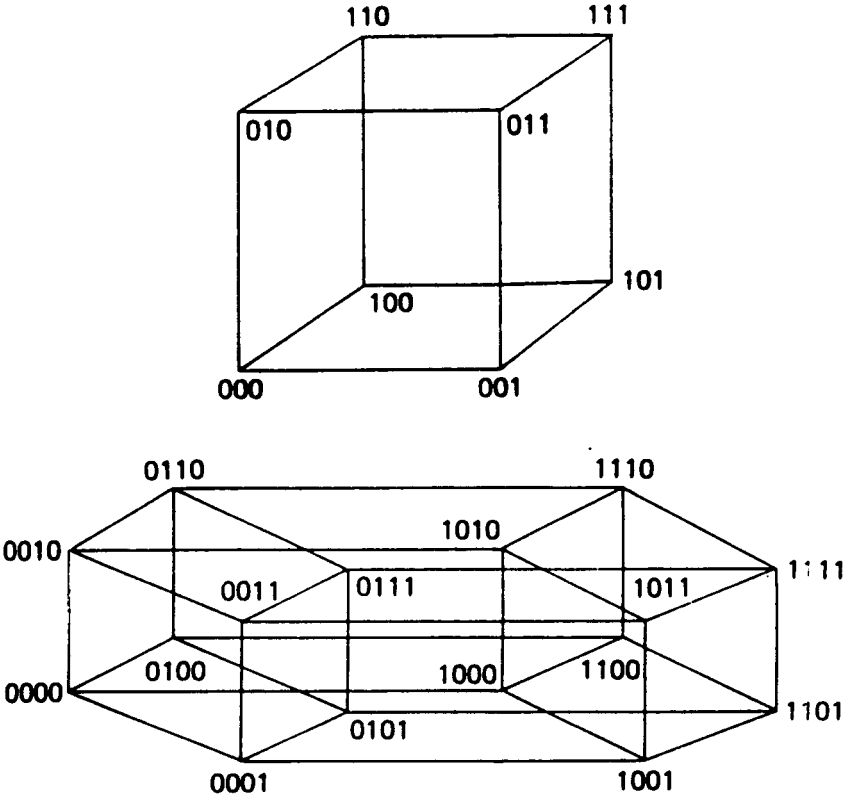


Figure 5.1. View of a 3-Dimensional and 4-Dimensional Cube with Vertices Numbered.

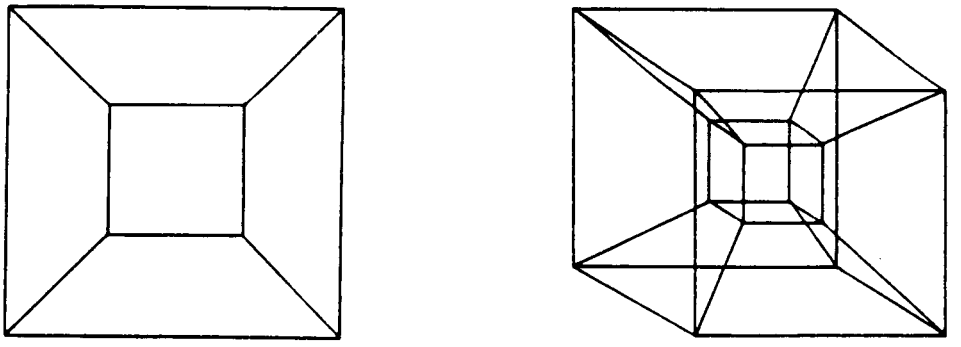


Figure 5.2. View of a 3-Dimensional and 4-Dimensional Cube Constructed from Lower-Dimension Cubes.

processes, typically providing the initialization for the application and the loading of the nodes with their individual programs. The host also communicates with the nodes via messages. The part of the application that is the node program executes in each node's operating system environment and runs concurrently. Typically these programs do calculations and exchange data via messages with other node processes and host processes [29].

5.2.1 The Host Program

The host program attends to the administrative functions of the simulation. These tasks include:

- loading the node processors with the node program
- coordinating the garbage collection of checkpoints
- determining the completion of the simulation

At appointed times, the host program will poll each node for its local virtual time and the number of checkpoints it has taken. If the host determines that the simulation has proceeded beyond a certain global virtual time (see Section 3.4.5), it will issue a garbage-collection request to the nodes.

The majority of the host's processing time is spent in idle, waiting for end of simulation notifications from the nodes. When this condition occurs and no rollback notifications are in transit, the host requests all nodes to write their data to an output file.

5.2.2 The Node Program

The algorithm for the node program is shown in Figure 5.3. Messages sent and received by the node are asynchronous, and are read at the beginning of each simulated time step. Processing of messages from nodes follows the rules governing the Time Warp mechanism, outlined in Chapter 3. Messages sent by the host are either queries for the local clock time or a request for the garbage-collection of old checkpoints.

Checkpointing of the node's state is done between the processing of internally generated events and externally generated events. This and the additional requirement of checkpointing immediately

```
Node_program {
    Read circuit description from input file
    Calculate portion of circuit to simulate (see section 5.3)
    Do until simulation ends {
        Check message queue and process message (see section 3.4)
        Process all events on eventlist at current time that were generated by internal
            nodes
        Send output changes and queue any internally generated events
        Checkpoint system according to scheduling discipline
        Process all events on eventlist at current time that were generated by external
            nodes
        Send output changes and queue any internally generated events
        Increment time
    }
    Notify host of completion
}
```

Figure 5.3. Algorithm for Node Program.

following a rollback will ensure that the system does not deadlock. It is expected that the likelihood of a rollback is greatest immediately following the occurrence of an externally generated event and will decrease as the time since the occurrence increases. Checkpoints are taken 2^N time steps after the (N-1) checkpoint following a rollback. This strategy proceeds until a maximum of 100 steps is reached between successive checkpoints; subsequently, checkpoints are then taken every 100 time steps. It should be noted, however, that the assumption of rollbacks exhibiting temporal and spatial localities has not yet been established theoretically or experimentally.

5.3 Assignment of Groups

The problem of finding an optimal solution for the assignment of groups for processing by nodes on the hypercube is an NP-complete problem. As a result, a much simpler and more scalable solution to the problem was implemented. Nodes are assigned groups in order of their node number, with node 0 initially assigned the group with the largest total fanin and fanout connections. An unassigned group is assigned for processing by node N if, among the remaining unassigned groups, it has the largest total number of fanin and fanout connections to all of N's adjacent neighbors that have already been assigned groups. The algorithm for assignment of groups to processors is given in Figure 5.4. Figure 5.5 shows the effects of the Placement_of_groups algorithm on a network of groups.

```
Placement of groups {
    Assign group with largest total fanin and fanout connection to processor 0
    For i = 1 to processor number {
        Group that is assigned to processor i: An unassigned group with the largest
            total fanin and fanout connections to all adjacent processors that have a
            processor number less than i
    }
}
```

Figure 5.4. Assignment of Groups to Processors.

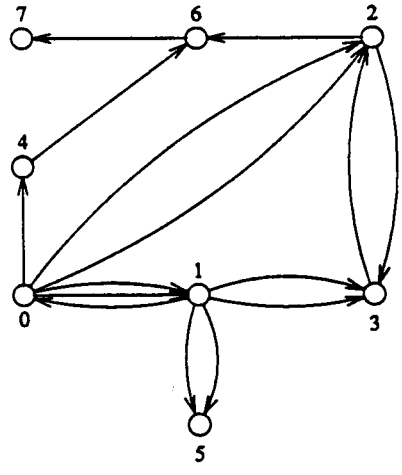


Figure 5.5. Example of the Assignment of Groups to Processors for a Sample Network.

CHAPTER 6

RESULTS AND CONCLUSIONS

6.1 Experimental Circuit

A set of experiments was designed to test the performance of the parallel simulation on the Intel *i*PSC/2 hypercube. An 8-bit ripple carry adder was chosen as the test circuit. The delay time for the propagation of value of the initial carry to the last stage in the adder circuit is proportional to the size of the adder; therefore, the ripple carry adder is an excellent choice as a test circuit to perform worst-case analysis of the parallel simulation. The general block diagram of an n-bit adder is shown in Figure 6.1 and the MOS transistor realization of an arbitrary full adder in the circuit is shown in Figure 6.2.

Identical simulations were run with a varying number of processors to detect the *speedup* of the simulation. Speedup is defined to be

$$S_p = t_N / t_1$$

where t_N is the time taken to run the simulation on N processors.

6.2 Results

The performance of the test circuit is given below in Table 6.1.

Table 6.1. Performance of Test Circuit with Timing Wheel Size = 256.

N	$t_N(\mu s)$	Speedup
1	3923	1.00
2	9717	0.40
4	10985	0.36
8	6818	0.58
16	6083	0.64

The results are not encouraging. One possible explanation is that checkpointing of the system and reclaiming of the checkpoint data upon resynchronization are absorbing a large percentage of the

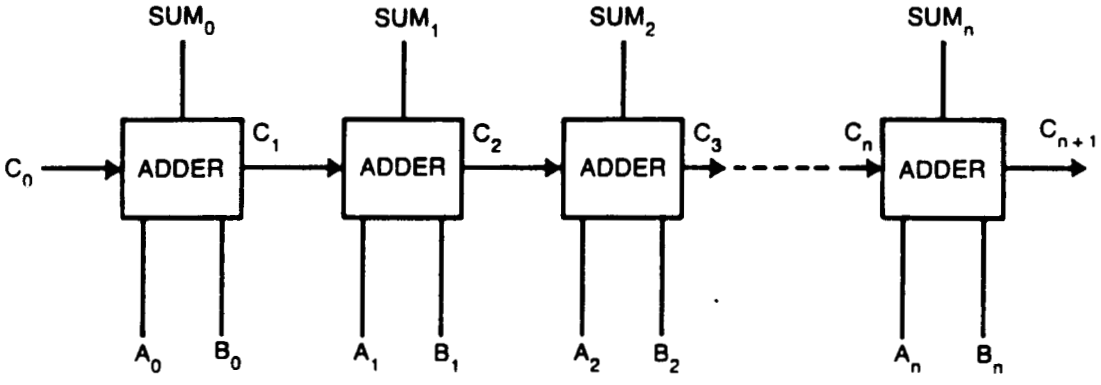


Figure 6.1. General Block Diagram of an N-bit Adder.

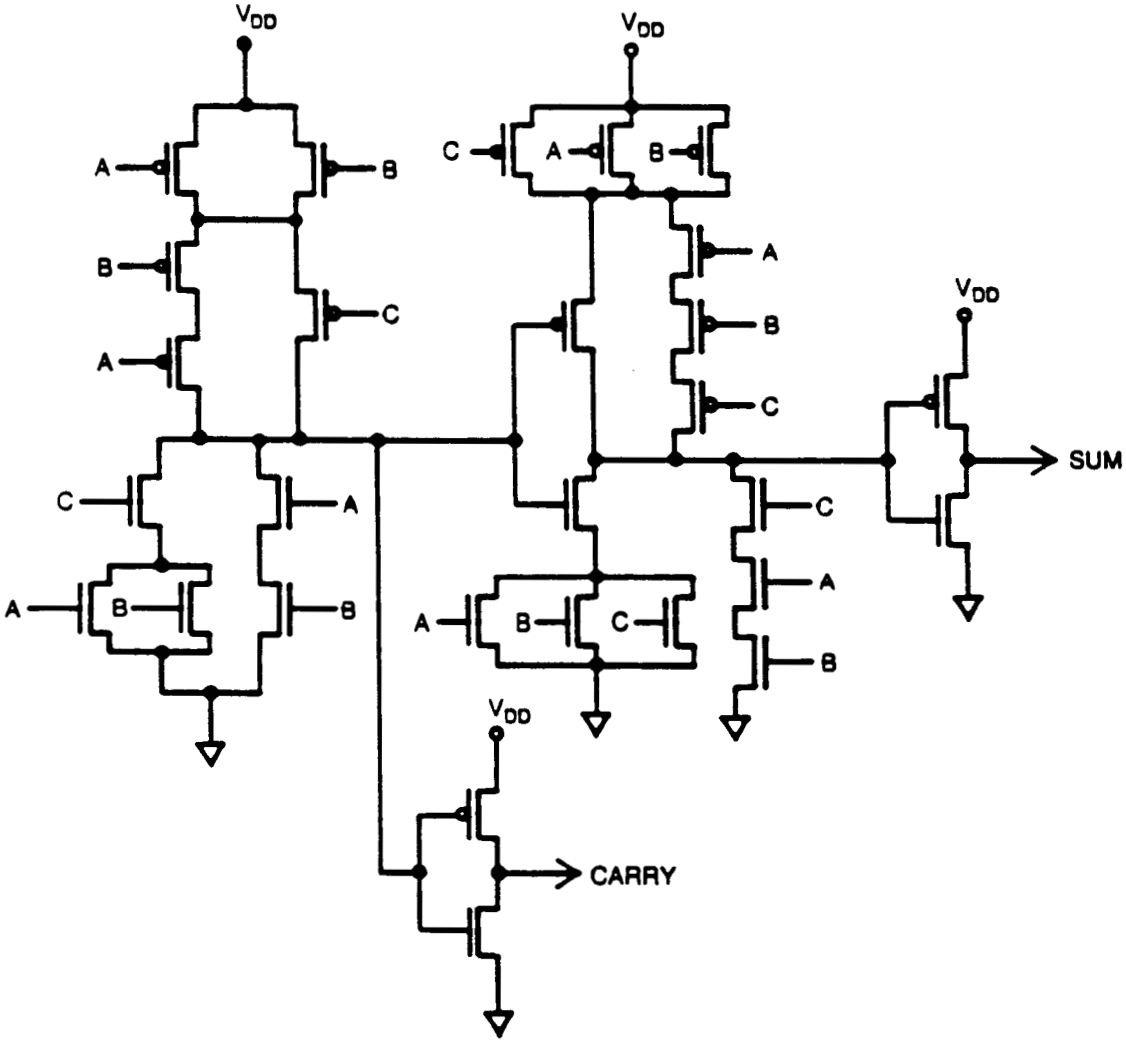


Figure 6.2. MOS Transistor Realization of a Full Adder.

simulation time. A solution to this problem would be to decrease the *timing wheel* size, that is, the number of time steps in the future that events can be stored on the eventlist without having to appear on the overflow list. By empirically varying the eventlist size, a value of 8 produced faster simulation times, listed in Table 6.2.

Table 6.2. Performance of Test Circuit with Timing Wheel Size = 8.

N	$t_N(\mu s)$	Speedup
1	2087	1.00
2	3463	0.60
4	2008	1.03
8	1148	1.82
16	1020	2.05

These results are considerably better; however, no speedup is achieved over one processor until the simulation is run on 8 or 16 processors. Clearly, additional side effects of the checkpointing mechanism have reduced the effects of parallelizing the sequential simulation.

A closer examination reveals that the frequency of checkpoints during a simulation vector can vary greatly based upon the number of rollbacks that occur. A typical situation occurs upon processing of a new vector: a rollback will occur based upon an incorrect assumption of the value of the carry bit. As the simulation proceeds, a minimum of 8 checkpoints will occur based on the checkpointing strategy of Section 5.2.2. However, since a one-processor configuration of the simulation will begin to checkpoint at a frequency of 2^X until a maximum of 100 time steps, an average of only 1.5 checkpoints will occur for each test vector beyond the first.

To test this theory, the history mechanism was changed to checkpoint after every time step, ensuring an equal number of checkpoints for each simulation. The results are presented in Table 6.3.

Table 6.3. Performance of Test Circuit with Checkpoint After Each Time Step.

N	$t_N(\mu s)$	Speedup
1	63948	1.00
2	52097	1.22
4	35949	1.78
8	21406	2.99
16	11895	5.38

The speedup results listed in Table 6.3 are significantly better; however, it should be noted that the simulation took 10 times longer for 16 processors with this method than it did with the history mechanism checkpointing on an exponential basis.

6.3 Conclusions

Several types of overhead exist in the Time Warp mechanism that indicate it might be prohibitive to use for parallel circuit simulation. From the results of the simulation experiments in Section 6.2, it is obvious that the checkpointing mechanism has very large space and time requirements. When the number of checkpoints is not equal among the simulations run on varying processors, it will be this factor that dominates in determining whether it is possible to achieve a speedup. Jefferson, in fact, has used the strategy of checkpointing at every time step for his benchmark tests. For circuit simulation, however, this method of checkpointing is not plausible for achieving quicker results as the actual run time of the simulation increases multifold.

Communication overhead exists when rollback occurs and triggers further rollbacks. Even when communication resources are plentiful, excessive computation can drain the computational resources of the processors, as every message requires some minimal amount of processing. It is imperative then that the partitioning of the test circuit will minimize the conditions that prompt rollback to occur. Arnold [30] requires that circuits are hand partitioned and in addition, has used carry lookahead logic in his test circuit to minimize the rollback phenomenon. His parallel simulator, PRSIM, has met with some success for this method, achieving a speedup of approximately 4 on 6 processors. However, the method of hand partitioning of circuits is not acceptable for large VLSI circuits, and furthermore, it is not clear how well PRSIM performs in worst-case scenarios.

Finally, there may also be a computational overhead due to invalid computations. If a processor receives an invalidation message but cannot interrupt the computations it is performing, then this overhead must also be taken into account. It is hoped, however, that the time lost for the completion of computations at a given time step does not outweigh the effects of the rollback mechanism.

6.4 Areas for Further Research

Solutions for these problems entail further research in the areas of checkpointing and partitioning of circuits. A possible solution for rapidly checkpointing a system may require the addition of dedicated hardware to create and restore checkpoints to the system. Dynamic partitioning or statistical-based partitioning of a circuit may help alleviate the problems of cascading rollbacks. Both of these issues must be addressed in the rollback mechanism in order for it to be useful for parallel circuit simulation.

In addition, other methods for parallel simulation should be explored. It may be that the method of using rollback for parallel simulation is not effective. In this case, methods such as the one proposed by Chandy and Misra should be explored.

LIST OF REFERENCES

[1] M. Denneau, "The Yorktown Simulation Engine," *Proceedings of the 19th Design Automation Conference*, pp. 55-59, 1982.

[2] M. Abramovici, Y. Levendel, and P. R. Menon, "A logic simulation machine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 82-93, 1983.

[3] J. Arnold and C. J. Terman, "A multiprocessor implementation of a logic-level timing simulator," *IEEE Transactions on Computer-Aided Design*, pp. 116-118, 1985.

[4] C. J. Terman, "RSIM - A Logic-level Timing Simulator," *International Conference on Computer Design*, pp. 437-440, 1983.

[5] D. R. Jefferson, "Virtual Time," *ACM transactions on Programming Languages and Systems*, pp. 404-425, 1985.

[6] L. Nagel, SPICE2: "A Computer Program to Simulate Semiconductor Circuits," *University of California at Berkeley, Memo ERL-M520*, 1975.

[7] A. R. Newton, D. O. Pederson, and A. L. Sangiovanni-Vincentelli, "Design aids for VLSI: a perspective revisited," *IEEE Design and Test*, pp. 106-115, 1985.

[8] J. Kleckner, R. Saleh, and A. R. Newton, "Electrical Consistency in Schematic Simulation," *IEEE International Conference on Circuits and Computers*, pp. 30-34, 1984.

[9] E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli, "The waveform relaxation method for time domain analysis of large-scale integrated circuits," *IEEE Transactions on Computer-Aided Design for Integrated Circuits*, pp. 131-145, 1982.

[10] N. Weste and K. Eshraghain, *Principles of CMOS VLSI Design*. Reading, Massachusetts: Addison Wesley, pp. 255-256, 1985

[11] S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "Logic circuit simulator," *The Bell System Technical Journal*, pp. 1451-1476, 1974.

[12] R. E. Bryant, "MOSSIM: A Switch-Level Simulator for MOS LSI," *Proceedings of the 18th Design Automation Conference*, pp. 786-790, 1981.

[13] C. J. Terman, "Simulation Tools for Digital LSI Design," *MIT Laboratory for Computer Science, TR-304*, pp. 1-84, 1983.

[14] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, pp. 198-206, 1981.

[15] J. Peacock, J. Wong, and E. Manning, "Distributed simulation using a network of processors," *Computer Networks*, pp. 44-56, 1979.

[16] V. Holmes, *Parallel Algorithms for Multiple Processor Architectures*. Ph. D. dissertation, Department of Computer Science, University of Texas at Austin, 1978.

[17] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation*. Englewood Cliffs, New Jersey: Prentice Hall, pp. 1-108, 1989.

[18] M. Seethalakshmi, *Performance Analysis of Distributed Simulation*. M. S. thesis, Department of Computer Science, University of Texas at Austin, 1978.

[19] O. Berry, *Performance Evaluation of the Time Warp Distributed Simulation Mechanism*. Ph. D. dissertation, Department of Computer Science, University of Southern California, 1986.

[20] A. Gafni, *Space Management and Cancellation Mechanisms for Time Warp*. Ph. D. dissertation, Department of Computer Science, University of Southern California, 1985

[21] D. Jefferson, "Implementation of Time Warp on the Caltech Hypercube," *Proceedings of the SCS Distributed Simulation Conference*, pp. 70-74, 1985.

[22] D. Jefferson, "The Status of the Time Warp Operating System," *ACM Transactions on Programming Languages and Systems*, pp. 738-742, 1988.

[23] M. Davon, *Partitioning Digital LSI Circuits for Parallel Simulation*. B. S. thesis, Department of Computer Science, Massachusetts Institute of Technology, 1986.

[24] S. Sahni, "NP-complete approximation problems," *Journal of ACM*, pp. 555-565, 1976.

[25] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, pp. 291-307, 1970.

[26] S. Smith and B. Underwood, "An Analysis of Several Approaches to Circuit Partitioning for Parallel Logic Simulation," *International Conference on Computer Design*, pp. 664-667, 1987.

[27] Y. Levendel, P. Menon, and S. H. Patel, "Special purpose computer for logic simulation using distributed processing," *The Bell System Technical Journal*, pp. 2873-2909, 1982.

[28] M. Heath, "The Hypercube: A Tutorial Overview," *Proceedings of the SIAM Conference on the Hypercube*, pp. 7-10, 1984.

[29] *iPSC System Overview*, Intel Corporation, 1986.

[30] J. Arnold, "Parallel Simulation of Digital LSI Circuits," *MIT Laboratory for Computer Science, TR-333*, pp. 53-62, 1985.