MIT/LCS/TR-333

# PARALLEL SIMULATION OF DIGITAL LSI CIRCUITS

Jeffrey M. Arnold

February 1985

MIT/LCS/TR-333

# PARALLEL SIMULATION OF DIGITAL LSI CIRCUITS

Jeffrey M. Arnold

# Parallel Simulation of Digital LSI Circuits

by

Jeffrey M. Arnold

© Massachusetts Institute of Technology, 1985

February 1985

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-83-K-0125.

Parallel Simulation of Digital LSI Circuits

by

Jeffrey M. Arnold

Submitted to the Department of Electrical Enginerring and Computer Science
on February 8, 1985 in partial fulfillment of the requirements for
the degree of Master of Science

*Abstract*

Integrated circuit technology has been advancing at a phenomenal rate over the last several years, and promises to continue to do so. If circuit design is to keep pace with fabrication technology, radically new approaches to computer-aided design will be necessary. One appealing approach is general purpose parallel processing. This thesis explores the issues involved in developing a framework for circuit simulation which exploits the locality exhibited by circuit operation to achieve a high degree of parallelism. This framework maps the topology of the circuit onto the multiprocessor, assigning the simulation of individual partitions to separate processors. A new form of synchronization is developed, based upon a history maintenance and roll back strategy. The circuit simulator PRSIM was designed and implemented to determine the efficacy of this approach. The results of several preliminary experiments are reported, along with an analysis of the behavior of PRSIM.

Name and Title of Thesis Supervisor:

Christopher J. Terman,
Assistant Professor of Computer Science and Engineering

Key Words and Phrases:

circuit simulation, parallel processing

Acknowledgements

I would like to thank my thesis supervisor, Chris Terman, for the inspiration and encouragement which made this thesis possible.

I would also like to thank everyone at RTS for the rich and stimulating environment they create, and especially Steve Ward, without whom RTS just wouldn't be. The following people deserve special thanks for everything they have contributed to my education:

Dave Goddeau

Bert Halstead

Rae McLellan

Al Mok

Jon Sieber

Tom Sterling

Rich Zippel

I would like to thank my parents and my entire family for their love and support over the years.

Finally, my deepest thanks to Maripat Corr for making everything worthwhile.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-83-K-0125.

# Table of Contents

**Chapter I Introduction** . . . . . 13

1.1. Overview . . . . . 13

1.2. Chapter Outline . . . . . 15

**Chapter II Parallel Simulation** . . . . . 17

2.1. A Framework for Parallel Simulation . . . . . 18

2.2. Synchronization . . . . . 20

2.2.1. Precedence Constraints . . . . . 20

2.2.2. Input Buffering . . . . . 21

2.2.3. Roll Back Synchronization . . . . . 22

2.2.4. Consistency Across Roll Back . . . . . 24

2.2.5. Checkpointing . . . . . 27

2.3. Partitioning . . . . . 28

2.4. Summary . . . . . 30

2.5. Related Work . . . . . 31
2.5.1. MSPLICE . . . . . 31
2.5.2. Virtual Time . . . . . 32

**Chapter III Implementation** . . . . . 35

3.1. Foundations . . . . . 35
3.1.1. The RSIM Circuit Simulator . . . . . 35
3.1.2. The Concert Multiprocessor . . . . . 36
3.2. The Organization of PRSIM . . . . . 38
3.2.1. The Prepass Phase . . . . . 39
3.2.2. The Coordinator . . . . . 40
3.2.3. The Simulation Slave . . . . . 42
3.3. Communication . . . . . 43
3.4. History Mechanism . . . . . 45
3.4.1. Simulation State Information . . . . . 46
3.4.2. Checkpoint Strategy . . . . . 47
3.5. Roll Back Mechanism . . . . . 48
3.6. Summary . . . . . 50

**Chapter IV Results** . . . . . 53

4.1. Overall Performance . . . . . 53
4.1.1. Experimental Procedure . . . . . 53
4.1.2. Results . . . . . 55
4.2. Profiling Results . . . . . 57
4.2.1. Experimental Procedure . . . . . 58
4.2.2. Results . . . . . 59
4.3. Discussion . . . . . 60

**Chapter V Conclusion** . . . . . 63

5.1. Summary . . . . . 63

5.2. Directions for Future Research . . . . . 65

5.3. Conclusion . . . . . 66

**Appendix A PRSIM Messages** . . . . . 69

**Appendix B History Implementation** . . . . . 75

**Appendix C Raw Performance Data** . . . . . 85

**Appendix D Profiling Data** . . . . . 87

**References** . . . . . 97

# Table of Illustrations

Figure 2.1. Partitioning a Network . . . . . 19
Figure 2.2. Data Dependence Between Two Partitions . . . . . 21
Figure 2.3. Data Dependence With Feedback . . . . . 21
Figure 2.4. Input Buffering Between Partitions . . . . . 22
Figure 2.5. Simulation Before Roll Back . . . . . 23
Figure 2.6. Simulation After Roll Back of Partition *A* . . . . . 24
Figure 2.7. Roll Back Notification . . . . . 25
Figure 2.8. Convergence Problem in the Presence of Feedback . . . . . 26
Figure 2.9. Data Path Floor Plan . . . . . 29
Figure 3.1. The Concert Multiprocessor . . . . . 37
Figure 3.2. Structure of PRSIM . . . . . 39
Figure 3.3. Simulation Control Loop . . . . . 42
Figure 3.4. PRSIM Message Structure . . . . . 43
Figure 3.5. Checkpointing the Network State . . . . . 47
Figure 3.6. Checkpoint Distribution . . . . . 47
Figure 3.7. Roll Back Procedure . . . . . 48

Figure 3.8. Response to Roll Back Notification . . . . . 50
Figure 4.1. 4-Bit Slice of Adder Circuit . . . . . 54
Figure 4.2. Adder Partitioning . . . . . 55
Table 4.3. Raw Performance Results in Events/Second . . . . . 56
Table 4.4. Simulation Efficiency and Speedup Factor . . . . . 56
Figure 4.5. Speedup Factor Versus Number of Partitions . . . . . 57
Table 4.6. Idle Time per Partition . . . . . 59
Table 4.7. Breakdown of Time Spent by Function . . . . . 60

Chapter I

# Introduction

An important component of any design process is a mechanism for incrementally checking the validity of design decisions and the interactions among those decisions. There must be a feedback path from the partially completed design back to the designer, allowing the designer to find and correct mistakes before fabrication. In modern digital circuit design, this feedback path is often provided by computer-aided simulation. However, in recent years integrated circuit technology has been advancing very rapidly. It is now possible to build chips containing more than 500,000 transistors. The current generation of simulation tools is already stretched to the limit, and will soon prove incapable of meeting this increase in demand. If circuit design is to keep pace with technology, radically new approaches to simulation will be necessary. One promising approach is to depart from the von Neumann style of computation and take advantage of recent advances in the field of parallel processing to build fast, scalable simulation tools.

## 1.1. Overview

In digital circuit design, the feedback path from a partially completed design back

to the designer is typically provided by computer-aided simulation. Historically, there have been two general approaches to circuit simulation: analytical and functional. Analytical simulators, such as SPICE, use detailed, non-linear models of circuit components drawn from fundamental physical principles, and solve the resulting set of ordinary differential equations using sparse matrix methods [12]. Because of this level of detail, analytical simulators tend to be computationally expensive, and so are limited in practice to the simulation of relatively small circuits (a few tens or hundreds of transistors). More recently, a number of algorithms have been developed to substantially improve the performance of circuit analysis programs. These include table lookup methods, such as those used in MOTIS [5], and iterated relaxation methods, such as those employed by SPLICE [18] and RELAX [13]. Although these newer techniques offer more than an order of magnitude performance improvement over the sparse matrix approach, they still cannot economically simulate one entire chip.

At the opposite end of the spectrum from circuit analysis are functional simulators, such as LAMP [4] and MOSSIM [3], which combine very simple models of circuit components, *e.g.*, gates or switches, with efficient event based simulation algorithms. This class of simulation tool is very useful for determining logical correctness, but offers no timing information. In the past few years, a third approach has emerged which tries to find a middle ground between analytical and functional simulation. Examples of this approach include the timing analyzers CRYSTAL [14] and TV [9], and the circuit simulator RSIM [19]. Each of these tools uses simple linear models of the electrical characteristics of the components to predict the timing behavior of a circuit. These tools permit one to obtain timing information on circuits of tens of thousands of devices, at the expense of some accuracy. Unfortunately, they are also reaching the limits of their capacities.

There are several approaches to solving the problem of capacity limitations. The first, and most obvious, solution is to vectorize the old algorithms to run on faster machines, such as the Cray and the CDC Cyber. The second approach is to develop

new, faster algorithms, such as the relaxation based schemes mentioned earlier. Another approach which has gained favor in certain circles is the development of special purpose hardware which is capable of running one specific algorithm very fast. Examples of this approach are the simulation pipeline of Abramovici [1], and the Yorktown Simulation Engine, developed by IBM [15]. Unfortunately, these solutions tend to be very expensive and applicable to only a very limited class of problems.

General purpose parallel processing offers several advantages over these other approaches.

- *Scalability* – Simulation algorithms can be developed which are independent of the number of processors in the system. As the size of the circuit grows, the number of processors, and hence the performance of the simulation, can grow.
- *Flexibility* – The machine architecture is not tuned for one particular algorithm. Therefore, the same physical hardware can be pressed into service for a wide range of applications, extending the utility of the machine.
- *Portability* – The parallel algorithms developed need not be constrained to a particular machine architecture. Therefore, the same algorithms can be run on a wide variety of parallel systems, extending the utility of the algorithms.

This thesis explores the issues involved in developing a framework for circuit simulation which can utilize the advantages offered by general purpose parallel computation. The approach is based upon the observation that the locality of digital circuit operation, and the resulting independence of separate subcircuits, leads very naturally to a high degree of parallelism. The framework developed in this thesis attempts to reflect the inherent parallelism of the circuit in the structure of the simulator.

## 1.2. Chapter Outline

Chapter 2 presents a novel approach to digital circuit simulation. This chapter

begins by exploring the techniques for mapping the circuit under simulation onto the topology of a general purpose multiprocessor. The synchronization problems imposed by the resulting precedence constraints are then examined, and a unique solution based upon history maintenance and roll back is proposed. The problem of partitioning a circuit in a fashion conducive to this form of simulation is then addressed. Finally, related work in the field of parallel simulation is reviewed.

Chapter 3 presents the implementation of the simulator Parallel RSIM, or PRSIM. This chapter begins with background information on the RSIM simulation algorithm and the Concert multiprocessor on which PRSIM is built. The overall structure of PRSIM is presented, with particular concentration on interprocessor communication and the history maintenance and roll back synchronization mechanisms.

Chapter 4 presents experimental results obtained from PRSIM. A series of experiments were designed and run to determine the overall performance of PRSIM, and to develop a solid understanding of the various overhead costs in PRSIM. The results from these experiments are analyzed, and some conclusions are drawn.

Chapter 5 concludes the thesis with a summary of the work reported and suggestions for future research.

Chapter II

# Parallel Simulation

Digital circuit operation exhibits a high degree of locality. At the device level, there is locality in the operation of individual transistors. Each transistor operates in isolation, using only the information available at its terminal nodes. At a somewhat higher level, there is locality in the operation of combinational logic gates. The output behavior of a gate is strictly a function of its input values. At a still higher level, there is locality in the operation of functional modules. The instruction decode unit of a microprocessor has no knowledge of what is transpiring in the ALU. It merely performs some function upon its inputs to produce a set of outputs.

The locality property of circuit operation is reflected in the structure of many simulation algorithms. So called event based simulators exhibit a similar degree of locality. A switch level simulator determines the value of a node by examining the state of neighboring switches. This locality property of the simulation algorithm implies the simulation of constituent subcircuits is independent. The simulations of two logic gates separated in space are independent over short periods of time.

This independence property has several interesting implications for the design of parallel simulation tools. First, it promises to be unaffected by scale. The potential parallelism increases linearly with the size of the circuit to be simulated. Second, it implies homogeneity of processing. Each processor can run the same simulation code on its own piece of the circuit. Third, the circuit database can be distributed across the multiprocessor. This eliminates the potential bottleneck presented by a shared network database, and allows the simulator to take advantage of the natural structure of the circuit.

In this chapter a framework for circuit simulation is presented which takes advantage of the independence inherent in circuit operation to achieve a high degree of parallelism. The general strategy is to map the circuit onto the target multiprocessor such that the parallelism of the simulation reflects the parallelism of the circuit. The framework uses a simple message passing approach to communication. Interprocessor synchronization is based upon a novel history maintenance and roll back mechanism.

## 2.1. A Framework for Parallel Simulation

There are several desirable properties our framework should have. First, the resulting simulator must be scalable. As the number of devices in the circuits that we wish to simulate increases, the performance of the simulator must also increase. Therefore, the framework should be capable of scaling to an arbitrary number of processors. Second, the framework should be relatively independent of the simulation algorithm. We would like to be able to apply the same strategy to a wide range of tools, from low level MOS timing analyzers to high level architectural simulators. Third, to permit our scheme to run on a variety of general purpose parallel machines, we must make no special demands of the underlying processor architecture. In particular, to be capable of running on both tightly and loosely coupled multiprocessors, a simulator should impose as few restrictions as possible on the nature of the interprocessor communication mechanism. We would like to avoid relying upon shared memory and imposing limits on message latencies.

The strategy we shall follow is to map the circuit to be simulated onto the topology of the target multiprocessor. For simulation on an $n$ processor system, the circuit to be simulated is first broken into $n$ subcircuits, or *partitions*. Each partition is composed of one or more *atomic units*, e.g., gates or subnets. An atomic unit is the collection of local network information necessary for the simulation algorithm to determine the value of a circuit node. Each processor is then assigned the task of simulating one partition of the circuit. Figure 2.1 demonstrates graphically the decomposition of a network of atomic units into two partitions.

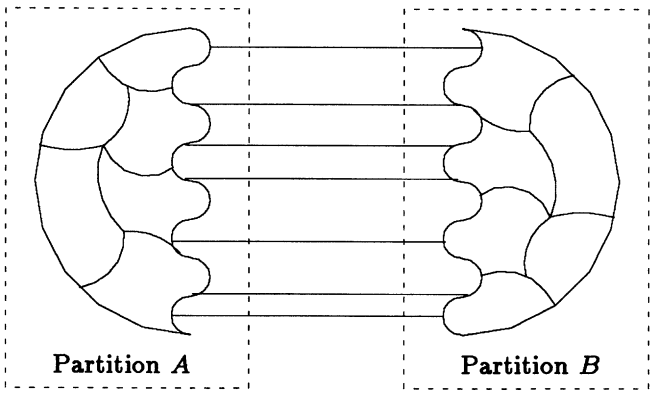


**Figure 2.1.** Partitioning a Network

The straight lines crossing the partition boundaries represent communication links between logically adjacent atomic units which have been placed in different partitions. In actual circuit operation, separate components communicate via the signals carried by electrical connections they have in common. Similarly, in simulation adjacent atomic units communicate only via the values of shared nodes. Therefore, the information which must be passed along the communication links consists of node values only. There is no need to share a common network database or pass non-local network information between partitions.

Communications issues tend to dominate the design of large digital circuits. Successful designs must constrain communication between submodules to meet routing and bandwidth requirements imposed by the technology. These constraints are similar

to those imposed by some multiprocessor architectures. Such constraints are often the source of performance limitations in parallel processing. Because the communication structure of the simulation in our framework is closely related to that of the actual circuit, our framework can easily utilize the natural modularity and optimizations in a circuit design to reduce interpartition, and hence interprocessor, communication.

In order to further reduce communication and to guarantee a consistent view of the state of the network across all processors, we shall enforce the restriction that the value of every node is determined by exactly one partition. Therefore, the links shown in Figure 2.1 will be unidirectional; a node may be either an input or an output of a partition, but never both. If more than one partition were allowed to drive a particular node, each partition would require information about the state of the other drivers to determine the correct value of the node. By eliminating the possibility of multiple drivers we eliminate the need for this non-local information and the extra communication required to arbitrate such an agreement.

This is not as serious a restriction as it first appears. In an MOS circuit, it implies all nodes connected through sources or drains of transistors, such as pullup and pulldown chains and pass transistor logic, must reside in the same partition. Since such structures are the components of higher level logic gates, it makes sense to keep them close together. The only difficulty arises from long busses with many drivers. This case results in a "bit slice" style of partitioning, where all of the drivers for one bit of the bus reside in the same partition, but different bits may reside in separate partitions. Since there tends to be relatively little communication from one bit to another, this restriction actually obeys the natural decomposition of digital circuits.

## 2.2. Synchronization

### 2.2.1 Precedence Constraints

A node shared between two partitions represents a precedence constraint. Enforcing this precedence constraint requires additional communication and can introduce

delay in a poorly balanced simulation. Consider the circuit in Figure 2.2. Let $T(A)$ be the current simulated time of partition $A$, and $T(B)$ be the current simulated time of partition $B$. For $B$ to compute the value of node $Y$ at $t_1$ it must determine the value of node $X$ at $t_1$. If at the point where $B$ requests the value of node $X$, $T(A) < t_1$ (*i.e.* $A$ is running slower than $B$), the request must be blocked until $T(A) \geq t_1$, potentially suspending the simulation of $B$. This interruption results from the need to synchronize the simulations of partitions A and B.

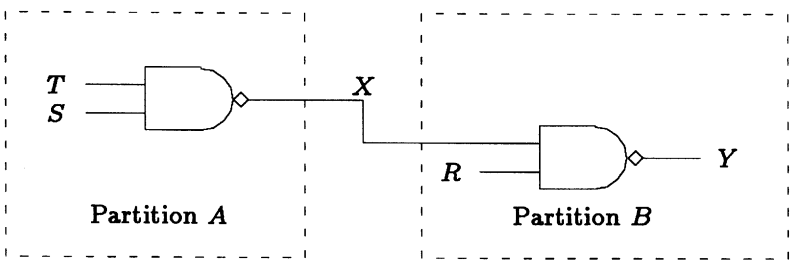


**Figure 2.2.** Data Dependence Between Two Partitions

The circular precedence constraint introduced by feedback between two (or more) partitions can result in a forced synchronization of the simulations. In Figure 2.3 feedback has been introduced into the previous example by connecting node $Y$ of partition $B$ to node $T$ of $A$. Each gate is assumed to have a delay of $\tau$ seconds. If $A$ has computed the value of $X$ at $T(A) = t_0$, $B$ is free to compute the value of $Y$ at $t_0 + \tau$. However, for $A$ to proceed to compute the value of $X$ at $t_0 + 2\tau$, it must wait until $T(B) \geq t_0 + \tau$, that is until $B$ has finished computing $Y$ at $t_0 + \tau$. The feedback has forced the two partitions into lock step, with each partition dependent upon a value computed during the previous time step of the other.

### 2.2.2 Input Buffering

These synchronization problems arise from the coupling between partitions introduced by shared nodes. With this in mind, the following observation can be made: If all partition inputs remained constant, there would be no precedence constraints to

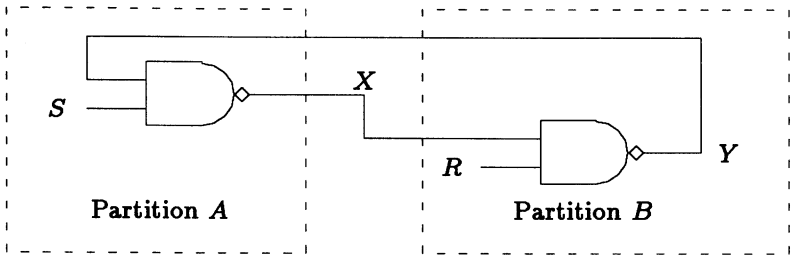


**Figure 2.3.** Data Dependence With Feedback

enforce. Each partition could be simulated independently of the others. This principle can be used to decouple partitions by introducing a level of buffering between each partition, as shown in Figure 2.4. Each partition maintains a buffer for each input node. Simulation is then allowed to proceed based upon the assumption that the currently buffered value of each input will remain valid indefinitely.

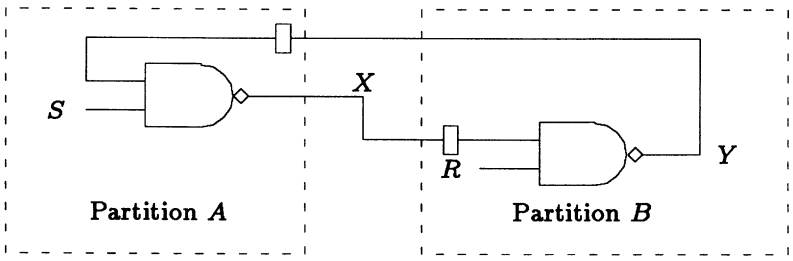


**Figure 2.4.** Input Buffering Between Partitions

When a partition changes the value of an output node, it informs all other partitions for which that node is an input. This is the basic form of interpartition communication. Changes in shared node values propagate from the driving partition to the receiving partitions. The information passed for a node change consists of a triple composed of the name of the node that changed, the new value of that node, and the simulated time the change took place. The receiving partitions use this information to update their input buffers, and, if necessary, correct their simulations.

### 2.2.3 Roll Back Synchronization

To maintain a consistent state of the network across the multiprocessor, some form

of synchronization is necessary. In the previous example, it is possible for partition $B$ to get sufficiently far ahead of $A$ that its assumption of constant inputs will result in incorrect simulation. Some form of correction is necessary. To this end, we employ a checkpointing and roll back strategy derived from the state restoration approach to fault tolerance in distributed systems [16] [17]. As the simulation progresses, a partition periodically stops what it is doing and takes a *checkpoint* of the current state of the simulation. This action is analogous to entering a recovery block in [16]. The checkpoint contains a record of all of the pieces of state in the partition: the value of every node, all pending events, and any state information kept by the simulation algorithm (*e.g.*, the current simulated time). From this checkpoint, the simulation of the partition can be completely restored to the current state at any future time, effectively rolling the simulation back to the time the checkpoint was taken. The set of saved checkpoints forms a complete history of the simulation path from the last resynchronization up to the current time.

When a partition receives an input change, one of two possible actions will occur. If the simulated time of the input change is greater than the current time, a new event representing the change is scheduled and simulation proceeds normally. However, if the simulated time of the input change is less than the current time, the simulation is "rolled back" to a point preceding the input change. This roll back operation is accomplished by looking back through the checkpoint history to find the most recent checkpoint taken prior to the scheduled time of the input change. The simulation state is then restored from that checkpoint, a new event is scheduled for the input change, and simulation is resumed from the new simulated time.

Figure 2.5 shows a partial history of the simulation of two partitions, $A$ and $B$. The time line represents the progression of simulated time. The "X" marks represent the times at which checkpoints were taken. The broken vertical line indicates a node change directed from one partition to another. The current time of each partition is shown by the corresponding marker.

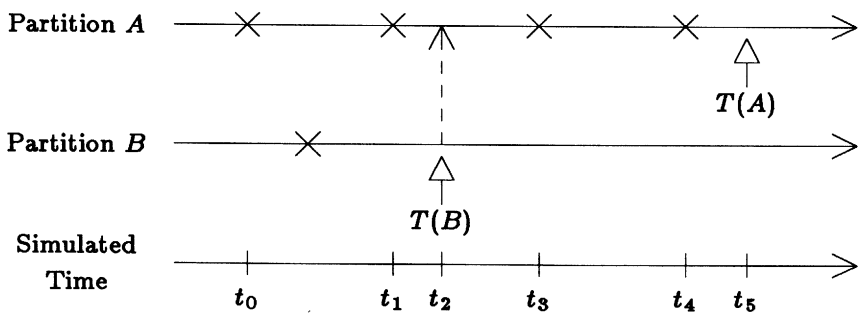


**Figure 2.5.** Simulation Before Roll Back

The snapshot shows the point when partition $B$ notifies $A$ that the value of a shared node changed at $t_2$. Upon receipt of the input change message, the simulation of $A$ is suspended and the checkpoint history is searched for the most recent checkpoint prior to $t_2$. The state of $A$ is then restored to time $t_1$ from the appropriate checkpoint. An event is scheduled for $t_2$ to record the change of the input node. The old simulation path beyond $t_1$ is now invalid, so all checkpoints taken after $t_1$ are thrown away. Partition $A$ is now completely restored to $t_1$ and simulation may continue. Figure 2.6 shows a snapshot of the simulation immediately following the completion of the roll back operation.

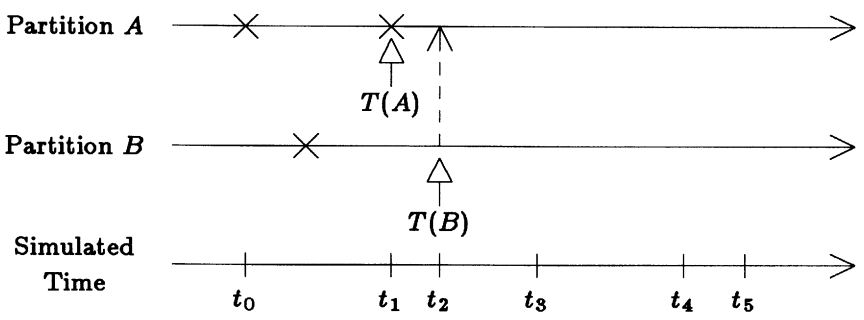


**Figure 2.6.** Simulation After Roll Back of Partition $A$

### 2.2.4 Consistency Across Roll Back

To maintain consistency across roll back, additional communication is required. Figure 2.7 shows the interactions among three partitions. At $t_3$ partition $C$ notifies $B$

that a shared node has changed value. Since $T(B) > t_3$, $B$ is forced to roll back to the most recent checkpoint prior to $t_3$, which is at $t_0$. The node change from $C$ to $B$ does not directly effect $A$. However, since $B$ will embark upon a new simulation path from $t_0$, the input change $B$ sent to $A$ at $t_2$ will be invalid. To ensure the consistency of $A$, a *roll back notification* message is introduced. Upon rolling back, $B$ sends $A$ a roll back notification message informing it that any input changes from $B$ more recent than $t_0$ must be invalidated. This does not necessarily force $A$ to roll back. If $T(A) < t_2$, the time of the earliest input change from $B$ more recent than $t_0$, $A$ need only flush the input change at $t_2$. If $T(A) > t_2$, $A$ would be forced to roll back to a point prior to $t_2$.

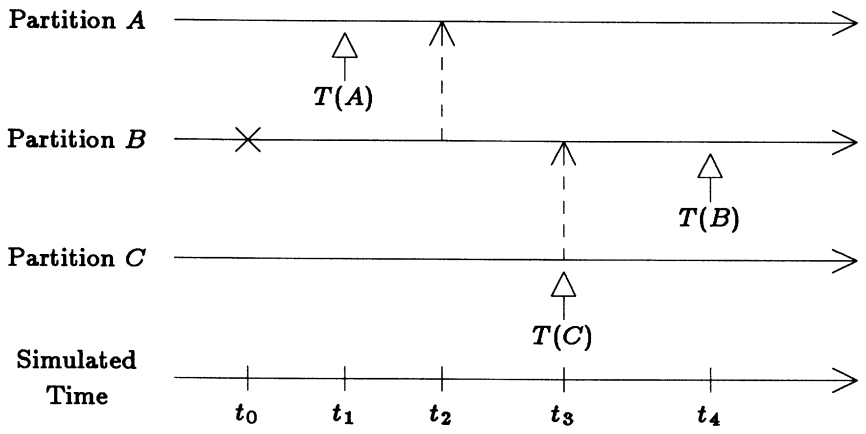


**Figure 2.7.** Roll Back Notification

The roll back notification procedure can be optimized if each partition maintains a history of output changes to implement a *change retraction* mechanism. At each time step, a partition checks the output history for the current simulated time. If, in a previous simulation path, an output change occurred which did not take place in the current path, a retraction is sent to all dependent partitions, and the output change is removed from the history. If the change did occur in the current path, no new change messages are necessary. Consider Figure 2.7. Since the change which forced $B$ to roll back occurred at $t_3$, $B$ will follow the same simulation path from $t_0$ to $t_3$, making the same node change at $t_2$. Therefore, $B$ need not resend this change to $A$. $A$ will not be

forced to roll back even if $T(A) > t_2$.

We must still address the problem of convergence in the presence of feedback. With the scheme outlined so far, it is possible for two partitions with a circular dependence to synchronize, with each partition repeatedly forcing the other to roll back. Figure 2.8 demonstrates this problem. When $B$ notifies $A$ of the change at $t_2$, $A$ will be forced to roll back to $t_0$. If $B$ progresses beyond $t_3$ before $A$ reaches $t_3$, $B$ will be forced to roll back to $t_1$. Once again, when $B$ reaches $t_2$, $A$ will be forced back to $t_0$, and the cycle repeats forever.

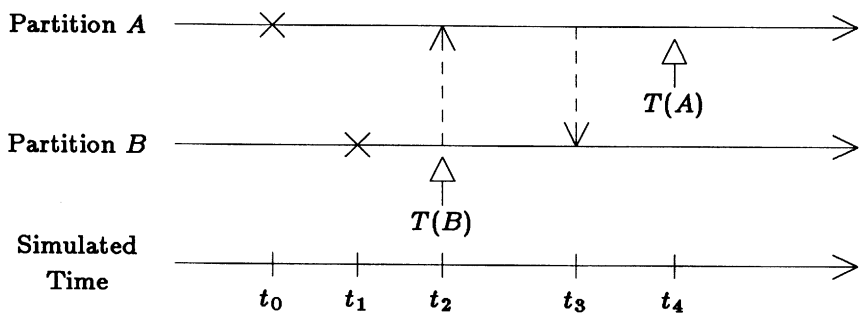


**Figure 2.8.** Convergence Problem in the Presence of Feedback

If $B$ had taken a checkpoint at $t$ such that $t_2 < t < t_3$, it would not have forced $A$ to roll back, and the cycle would have been avoided. However, if the changes occur simultaneously ($t_2 = t_3$), we are again faced with the infinite cycle. To solve this problem, we first make the following assertion about the nature of the simulation algorithm: *the elapsed simulated time between an input change and any resulting new events is non-zero.* This assertion can be made true by proper partitioning of the network. This restriction allows the simulation of a single time step to be sub-divided into two distinct phases:

1. the processing of all internally generated events queued for the current simulated time, including the propagation of output changes to other partitions;
2. the processing of all externally generated input changes queued for the

current simulated time.

This in turn permits us to take a checkpoint between the two phases of the simulation, after any output changes have been made and before any input changes have been processed. Returning to the example of Figure 2.8, if $B$ were to take a checkpoint at $t_2$, it could be rolled back safely without causing a further roll back in $A$, even in the limit of $t_2 = t_3$. Forward progress is assured if we can guarantee there will always be a checkpoint in the right place.

The convergence problem is related to the "domino effect" observed in distributed systems, where one failure can cause many interdependent processes to repeatedly roll back until they reach their initial state [16][17]. In the context of simulation we have shown that this problem arises from synchronization of precedence constraints imposed by the partitioning. Under these circumstances, the best that can be done, short of dynamically repartitioning to ease the constraints, is to guarantee convergence. This is done by subdividing the simulation of a single time step into two phases, and checkpointing between the phases.

### 2.2.5 Checkpointing

The checkpointing strategy must meet the following constraints: the checkpoint must contain all of the state necessary to completely restore the simulation; there must always be at least one consistent state to fall back to; and it must be possible to make forward progress in the event of unexpected synchronization. In addition to these constraints, there are some less important but still desirable properties a checkpoint strategy should have. For example, to prevent rolling back further than necessary, the simulation should be checkpointed frequently. In the limit, a checkpoint at every time step would eliminate redundant work. We would also like the checkpointing process to be as inexpensive in both space and time as possible. There is a tradeoff between the cost we are willing to pay when forced to roll back and the cost we are willing to pay for checkpointing overhead.

We expect the communication between partitions in a statically well-partitioned

circuit to be clustered in time, e.g., around clock edges. This implies the probability of receiving a node change is greatest immediately following a change, and decreases as the time since the last change increases. The probability of roll back should follow a similar pattern. Therefore, to reduce the amount of redundant simulation caused by rolling back, we would like to have a high density of checkpoints in the vicinity of communication clusters. If the dynamic balance of the partitioning is less than ideal, some of the partitions will simulate faster than others. In this case, the amount of redundant work forced upon the faster partitions by roll back is less critical, as they will still catch up to and overtake the slower partitions. Therefore, if the time since the last roll back is large, we can afford to reduce the density of checkpoints.

These observations have lead to a strategy of varying the frequency of checkpointing with time. Following each resynchronization and each roll back, a checkpoint is taken at every time step for the first several steps, thus ensuring forward progress as well as providing a high density of checkpoints. As the simulation progresses, the number of time steps between checkpoints is increased up to some maximum period. The longer the simulation runs without rolling back, the lower the checkpoint density, and hence the overhead, becomes. We have arbitrarily chosen to use an exponential decay function for the frequency until we have a better model of the probability distributions of interpartition communication.

## 2.3. Partitioning

The overall performance of the simulator is determined by two factors: processor utilization, and communication costs. Both of these factors are influenced by the manner in which the network is partitioned. To maximize processor utilization, the simulation load must be evenly distributed among the processors. This implies partitioning the circuit into pieces of roughly equal size and complexity. To minimize communication costs, the number of links between partitions should be minimized. There are a number of classical graph partitioning algorithms which address both of these criteria [10][11].

For example, consider the data path block diagram shown in Figure 2.9. A static analysis of this circuit shows most of the communication paths are horizontal, from left to right. Only in the carry chain of the ALU and in the shifter will there be any communication from bit to bit. A static min-cut algorithm would partition this circuit into horizontal slices, following the flow of information along each bit. One would expect this partitioning to result in an even load balance, with little interprocessor communication.

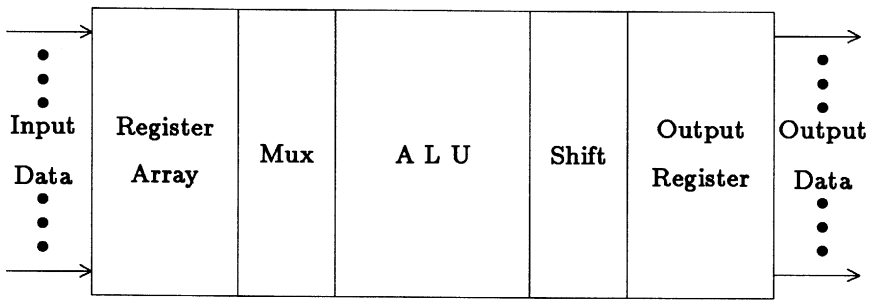


**Figure 2.9.** Data Path Floor Plan

Unfortunately, there are dynamic components to both processor utilization and communication with which static partitioning algorithms are unable to cope. For example, consider a 16-bit counter to be split into 4 partitions. A static min-cut algorithm would divide this circuit into four 4-bit slices, in the same manner as the data path above. Each partition would be exactly the same size, have only one input (the carry in) and one output (the carry out). At first glance, this would seem to be a fine partitioning. The dynamic behavior, however, will be quite poor. Both the simulation load and the communication decrease exponentially from the low order partition to the high order one, with the low order partition doing eight times the work of the high order one. A more effective partitioning would have placed bit 0 of the counter (the low order bit) in the first partition; bits 1 and 2 in the second partition; bits 3–6 in the third; and bits 7–15 in the last. The dynamic load would then be much more evenly distributed.

Clearly, a partitioning strategy based only upon the static structure of the circuit will not fare well under a wide range of applications. Some knowledge of the dynamic

behavior of the simulation is necessary. One approach would be to begin with a static partitioning, but dynamically repartition the network during the simulation by shuffling atomic units between processors to optimize the load balance and communication. This topic is beyond the scope of this thesis, and deserves future investigation.

## 2.4. Summary

In this chapter we have presented a framework for simulation which takes advantage of the parallelism inherent in digital circuit operation. We proposed a scheme in which the circuit to be simulated is partitioned onto the topology of the multiprocessor, with each processor responsible for the simulation of one partition. We discussed the problems of synchronization introduced by this approach, and developed a solution based upon a history maintenance and roll back mechanism. This solution was demonstrated to be sufficient to guarantee convergence in the presence of feedback. Finally, we discussed the importance of good partitioning, and showed that static graph partitioning algorithms may not be adequate.

We began this chapter by setting out three goals for a parallel simulation framework. Let us now see how close our proposed framework comes to those goals.

- The framework is scalable to a large number of processors. As the size of the circuit grows, we can increase the number of partitions, keeping the average size of the partitions constant. The factors which will probably limit the scalability will be the interprocessor communication mechanism (*e.g.*, bandwidth, congestion), and the effectiveness of the partitioning algorithm.
- The framework does impose some constraints upon the nature of the simulation algorithm. We require an event based simulator which exhibits a high degree of locality. A wide range of simulation tools will fit this description, but we exclude most low level circuit analysis programs, such as SPICE.
- The framework has few requirements of the underlying multiprocessor

architecture. The small amount of communication required makes it suitable for both tightly and loosely coupled systems. The overall performance should degrade gracefully with increasing message latencies.

## 2.5. Related Work

The problems of parallel simulation have received a great deal of attention recently. A number of the resulting research efforts have influenced the work reported in this thesis. Among the most influential have been the work on the MSPLICE parallel simulator and the Virtual Time model for distributed processing.

### 2.5.1 MSPLICE

MSPLICE is a multiprocessor implementation of a relaxation based circuit simulator [6]. The algorithm employed is known as *Iterated Timing Analysis*, and is based upon Newton-Raphson iteration to approximate the solution of the node equations which describe the circuit. It makes use of event driven, selective trace techniques similar to those employed by SPLICE to minimize the amount of computation required per time step of simulation [18].

The Iterated Timing Analysis method is extended for implementation on a multiprocessor by a "data partitioning" technique. The circuit to be simulated is divided into sub-circuits, with each sub-circuit represented by a separate nodal admittance matrix. Each sub-circuit is then allocated to a processor. Each processor, operating on the same time step, applies the ITA algorithm to each of its sub-circuits until convergence is reached. When every sub-circuit on every processor has converged, the simulation advances to the next time step. Synchronization is achieved through a global variable which represents the count of outstanding sub-circuit events for the current time step.

The approach to parallelism followed by MSPLICE is quite close to that of our proposed framework. Both schemes seek to exploit the parallelism inherent in the circuit through a data partitioning strategy: the circuit to be simulated is distributed across the multiprocessor, with each processor running the same algorithm on different data.

There are several important differences, though. The MSPLICE algorithm is necessarily synchronous, with all of the processors simulating the same time step. This has two important implications. First, the time required to simulate a particular time step is determined by the slowest partition. Second, additional communication is required to manipulate the global synchronization counter. Because of the nature of the relaxation method, MSPLICE does not have the same locality properties as our framework. The information necessary to compute the node values of a given sub-circuit is not necessarily local to a single processor. For each iteration, each processor must fetch the current values of all of the *fanin* nodes for each sub-circuit, and propagate events to all of the *fanout* nodes. The communication requirements of MSPLICE imply a dependence upon shared memory and a tightly coupled multiprocessor architecture, which we have tried to avoid.

### 2.5.2 Virtual Time

*Virtual Time* is a model for the organization of distributed systems which is based upon a lookahead and rollback mechanism for synchronization. In this model, processes coordinate their actions through an imaginary *Global Virtual Clock*. Messages transmitted from one process to another contain the virtual time the message is sent and the virtual time the message is to be received. If the *local virtual time* of the receiver is greater than the virtual time of an incoming message, the receiving process is rolled back to an earlier state [8].

The basic strategy of Virtual Time is quite close to that followed by our simulation framework presented earlier. Both propose the use of state restoration as a mechanism for the synchronization of parallel processes. The principal difference is that Virtual Time is proposed as a general model for all forms of distributed processing. We are only using the roll back synchronization in a very limited, very well characterized domain. This has several implications. First, we take advantage of knowledge about the context to strictly limit the amount of state information we must keep. The Virtual Time model requires saving the entire state of the process, including the stack and all

non-local variables, at every checkpoint. Second, we have organized the problem such that the amount of interprocessor communication is quite small. This in turn leads to relatively infrequent roll backs. Third, we are able to make assumptions about the distribution of the communication to reduce the frequency of checkpointing. It is not clear how frequently the state must be saved in the Virtual Time system. Fourth, by subdividing the simulation time step and carefully choosing the checkpoint strategy, we are able to guarantee the convergence of the simulation. The general convergence properties of Virtual Time are less well characterized. By taking advantage of the structure of the simulation algorithm, the history maintenance and roll back approach to synchronization becomes much more tractable.

Chapter III

# Implementation

It is all very well to theorize about parallel processing, but the best way to assess the efficacy of a new idea is to try it. A simulator based upon the parallel framework presented in Chapter Two was designed and built with the following goals:

- to determine whether the roll back approach to interprocessor synchronization can be made cost effective in the context of circuit simulation;
- to produce a fast, scalable circuit simulator capable of simulating the next generation of VLSI circuits efficiently.

This chapter discusses the details of the implementation of that simulator.

## 3.1. Foundations

Parallel RSIM, or PRSIM, is a distributed circuit simulator which employs the history and roll back mechanisms discussed in Chapter Two. As the name implies, PRSIM is based upon the RSIM algorithm of [19]. It is implemented on the Concert multiprocessor, developed at MIT [2][7].

### 3.1.1 The RSIM Circuit Simulator

RSIM is an event-driven, logic level simulator that incorporates a simple linear model of MOS transistors. In RSIM, MOS transistors are modeled as voltage controlled switches in series with fixed resistances, while transistor gates and interconnect are modeled as fixed capacitances. Standard *RC* network techniques are used to predict not only the final logic state of each node, but also their transition times. This relatively simple and efficient model provides the designer with information about the relative timing of signal changes in addition to the functional behavior of the circuit without paying the enormous computational costs of a full time domain analysis.

The electrical network in RSIM consists of nodes and transistors. Any MOS circuit can be naturally decomposed into subnets if one ignores gate connections; the resulting subnets each contain one or more nodes which are electrically connected through the sources or drains of transistors. The nodes connected to gates of devices in a subnet are the *inputs* of the subnet, and the nodes which are inputs of other subnets are the *outputs* of the subnet. Note that a node can be both an input and output of a single subnet.

Subnets are the atomic units of the simulation calculation; in general RSIM will recalculate the value of each node of a subnet if any input to the subnet changes. If, as a result of the recalculation, an output node changes value, an *event* is scheduled for the simulated time when the output is calculated to reach its new value. Processing an event entails recomputing node values for subnets that have the changing node as an input.

Internally, RSIM maintains a single event list where all unprocessed events are kept in order of their scheduled time. When a node changes value, all other nodes which are affected by that change are examined. For each affected node that changes value, the simulated time of the change is computed and an event is added to the event list in the appropriate place. The next event to be processed is then taken from the beginning of the list, and the cycle repeats itself. A simulation step is considered complete when

the event list is empty, *i.e.* when no more changes are pending.

### 3.1.2 The Concert Multiprocessor

*Concert* is a multiprocessor test bed designed to facilitate experimentation with parallel programs and programming languages. It is organized as a ring of clusters, with 4 to 8 Motorola MC68000 processors in each cluster, as shown in Figure 3.1. The processors in each cluster communicate via shared memory across a common bus, although each processor has a private, high speed path to a block of local memory. The clusters communicate via globally accessible memory across the *RingBus*. Each processor therefore sees a three level hierarchy of memory:

1. high speed memory accessible over the processor's private "back door" path (this memory is still accessible to other processors in the cluster via the shared bus);
2. slower, non-local cluster memory accessible over the shared cluster bus;
3. global memory, accessible only through the RingBus.

All three levels of the hierarchy are mapped into the address space of each processor. Therefore, the memory hierarchy can be treated transparently by the user program if it is convenient to do so. Note that non-global cluster memory is not accessible from the RingBus [2][7].

Over time, a large set of subroutine libraries have been developed for the Concert system. One such library, the Level 0 Message Passing library, implements a reliable message delivery system on top of the Concert shared memory system. For each processor there exists a message queue in global memory. To send a message, the L0 system copies the message body into global memory if it is not already there, and places a pointer to the top of the message body into the receiving processor's queue. To receive messages, the queue is polled on clock interrupts. Messages on the queue are removed and returned to the user program by a user-supplied interrupt handler. The L0 package also provides a set of functions for sending and receiving messages.

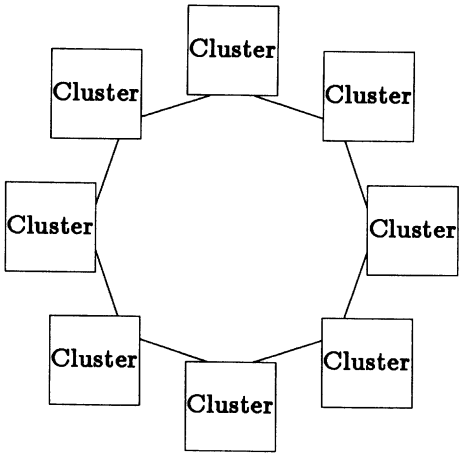


**Figure 3.1.** The Concert Multiprocessor

The original RSIM program used floating point arithmetic for the Thevenin and $RC$ calculations. Concert has no floating point hardware, so it was felt that rather than emulate the floating point arithmetic in software, it would be more efficient to use scaled fixed point arithmetic. A 32-bit integer can represent a range of roughly 9 decimal orders of magnitude, more than sufficient for the ranges of resistance, capacitance, and time found in contemporary MOS simulation. The actual ranges of the units used by PRSIM follow:

$$0.1\Omega \leq R \leq 100M\Omega$$

$$10^{-6}pF \leq C \leq 1000pF$$

$$0.1nS \leq t \leq 100mS$$

To represent the products and quotients of these units without loss of precision, a scaled arithmetic package using 64-bit intermediate results was written. The routine `RCMul(R, C)` computes the 64-bit product of a resistance and a capacitance, and then divides by a constant scale factor to produce a 32-bit time quantity. The routine `MulDiv(A, B, C)` multiplies any two 32-bit integers, and divides the 64-bit product by a third 32-bit integer to yield a 32-bit result. This is useful for the Thevenin resistance calculation. Finally, the routine `CvtCond(R)` converts a resistance to a conductance (and *vice versa*) by dividing its argument into a 64-bit constant to yield a scaled 32-bit

result.

## 3.2. The Organization of PRSIM

The PRSIM system consists of two phases: a prepass phase and a simulation phase. The prepass phase is responsible for partitioning the network to be simulated and for compiling the result into an efficient machine readable format. The simulation phase itself can be further broken down into a coordinating program and a simulation program. In an $n$ node multiprocessor, 1 processor is dedicated to the user interface and coordination functions, while the remaining $n-1$ processors do the actual simulation work. This organization is illustrated in Figure 3.2.

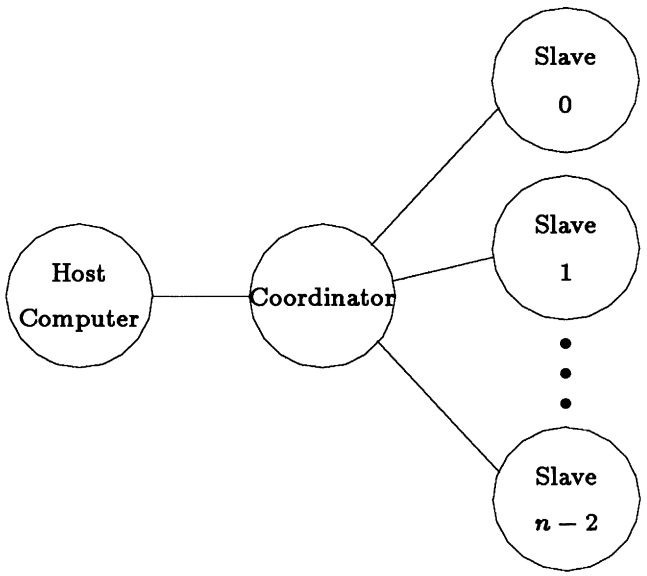


**Figure 3.2.** Structure of PRSIM

### 3.2.1 The Prepass Phase

The operation of PRSIM begins with the circuit to be simulated expressed in the lisp-like description language NET [20]. † In the NET description the user may also specify the desired partitioning of the circuit. From this high level description, the PRESIM

† At present, PRSIM has no automatic partitioning system. When such a mechanism is available, PRSIM will also be able to simulate a circuit extracted from a mask level description.

program, running on a conventional computer, first partitions the circuit into $n-1$ pieces based upon the user's specification and the constraints imposed by the parallel framework and the RSIM algorithm. Next, the dependencies between the partitions are determined and the mapping tables used by each partition and by the coordinator are constructed. Each output node of each partition is given a list of partitions for which that node is an input. Finally, $n$ binary files are produced, one for each partition and one for the coordinator.

### 3.2.2 The Coordinator

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

- loading the network files for each of the partitions from the host computer;
- running the user interface to the simulator, including getting and setting node values;
- starting, stopping, and resynchronizing the simulation.

The coordinator handles all input and output with the host computer. Upon initialization it searches out the active processors in the system and reads the coordinator file generated by PRESIM from the host to obtain the number of partitions to be simulated. For each circuit partition it assigns a processor from the active pool and passes it the name of the appropriate network database file. Each slave processor is then responsible for reading the appropriate file by sending read requests to the host through the coordinator.

PRSIM supports two different user interface languages: a simple line-at-a-time command interpreter for simple operations, and a lisp-like language for more elaborate control structures [20]. Through either of these interfaces the user may get and set node values, examine the network structure, and start or stop the simulation.

Each node in the circuit is identified by a globally unique identifier, or *node ID*, which is assigned during the prepass phase. The coordinator maintains a table of *node*

*entry* data structures, one for each node in the circuit. This table can be referenced in two different ways: indexed by global node ID, for mapping IDs into names for the user; and hashed on the ASCII name of a node, for mapping the user specified ASCII names into global node IDs. In addition to this two-way mapping, the node entry structure also identifies the partition responsible for driving the node and contains a list of partitions for which this node is an input. This information is used to permit the user to examine and set node values.

When the user requests the value of a particular node, the ASCII name provided by the user is first mapped into the corresponding node ID by the hash table. A message requesting the value of the node is sent to the partition responsible for computing that value. The partition then looks up the value of the node and sends back a reply message. When the user wishes to set the value of a node, the coordinator sends the driving partition a message containing the ID of the node, the new value for the node, and the simulated time of the change. No reply is necessary.

To start a simulation step, the coordinator first establishes user supplied input conditions by sending input change messages as necessary to the slave processors. When all of the input changes have been established, the coordinator starts the simulation by sending a STEP message containing the desired termination time to each slave processor. When each processor reaches the specified stop time, it sends a SETTLED message back to the coordinator and waits. Since a processor may be forced to roll back after it has reached the stop time, roll back notifications are sent to the coordinator as well. With this information, the coordinator keeps track of the state of the simulation of each partition. When it has determined that all of the slave processors have safely reached the stop time, the coordinator sends a RESYNC message to each slave to inform it that its old history is no longer needed and may be reclaimed.

In the current implementation the simulation is resynchronized only at the termination of each test vector. Since there is some overhead costs associated with starting and stopping the simulation, the longer the simulation is allowed to run asynchronously,

*i.e.*, the longer the test vector, the less significant the overhead cost will be. However, since checkpoint histories are only reclaimed at resynchronization time, the amount of storage devouted to checkpointing becomes the factor which limits the length of the test vectors. In future implementations, a mechanism for pruning old checkpoints together with automatic resynchronization initiated by the coordinator could be used to extend the length of the vectors.

### 3.2.3 The Simulation Slave

The simulation slave program is composed of three components: the simulation loop; the interprocessor communication mechanism; and the history and roll back synchronization mechanism. The simulation control loop is shown Figure 3.3. `CurTime` is the current simulated time of the partition, and `StopTime` is the termination time specified by the coordinator.

```
while CurTime ≤ StopTime
  { /* process events queued for CurTime */
    for each event scheduled for CurTime
        process event;
    send queued output changes;
    if time to checkpoint
        checkpoint();
    /* end of phase one */
    /* process inputs queued for CurTime */
    for each event scheduled for CurTime
        process input;
    /* end of phase two */
    CurTime = CurTime + 1;
  }
```

**Figure 3.3.** Simulation Control Loop

The processing of events proceeds as follows. For each event scheduled for `CurTime`, the event is removed from the list, the specified node change is made, and the effects are propagated through the partition. If the node specified in the event is an output, the event is added to the *output change list*. When all events scheduled for `CurTime`

have been processed, one *input change* message is constructed for each partition which is dependent upon one or more of the outputs in the list. Each message contains the value of `CurTime` and the ID and new value of each node in the list which is an input to the receiving partition. Once the input change messages have been sent, the output change list is cleared, completing the first phase of the simulation. At this point, if a sufficient period of time has elapsed since the last checkpoint, a new checkpoint is taken (see Section 3.4 for more detail).

The operation of the second phase of the simulation is similar. For each input change there is a data structure which contains the ID of the input node, the new value, and the simulated time of the change. These structures are kept in the doubly linked `InputList` sorted by simulated time. The `NextInput` pointer identifies the next input change to be processed. For each input change scheduled for `CurTime` the specified node change is made and the effects propagated through the network. After each change is processed, the `NextInput` pointer is advanced. The `InputList` remains intact.

By subdividing the simulation of a single time step into the two phases shown, and by checkpointing at the end of the first phase, any roll back will restore the simulation to the beginning of the second phase. Since the elapsed time between an input change and any resulting event is non-zero, the simulation will converge in the manner described in Chapter Two, although it may require several roll back operations.

## 3.3. Communication

There are two classes of interprocessor communication in the PRSIM system: administrative communication with the coordinator for such purposes as loading the partition data base and answering queries from the user; and interpartition communication required for sharing circuit nodes across multiple partitions. Both of these forms of communication make use of a low level message management system which itself is built upon the reliable message delivery protocol of the Concert Level 0 system.

Figure 3.4 shows the structure of a PRSIM message. The whole message consists

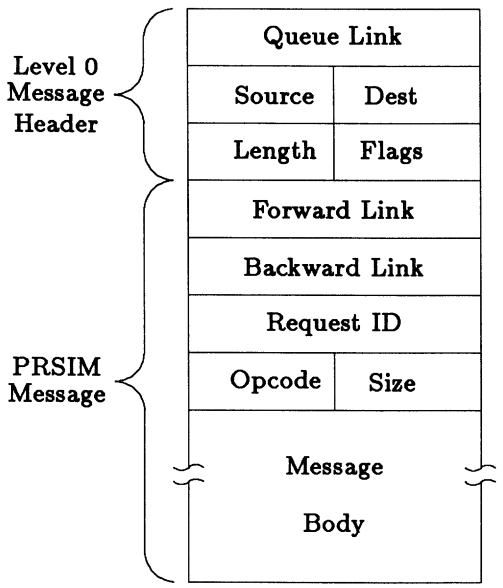


**Figure 3.4.** PRSIM Message Structure

of two components: a Level 0 header, which is used by the Concert Level 0 software, and the PRSIM message itself. The PRSIM message is further composed of a message header and a body. This header contains two links for the doubly linked active message list; a request ID for matching replies to synchronous requests; an opcode field which identifies the type of message; a size field which determines the length of the message; and finally the message body, which contains the data. Message bodies are a multiple of 16 bytes in length, up to a maximum of 1024 bytes. The body size of a message is determined when the buffer for the message is first allocated. When a message has finished its task, its buffer is returned to a free list managed by the sending processor, from which it may be reallocated later. To avoid searching one free list for a buffer of a certain length, there are 64 separate free lists, one for each possible message size. Messages of the same size are returned to the same free list. A complete list of PRSIM messages appears in Appendix A.

To send a message, a processor obtains a buffer of the appropriate size from the free list, allocating a new one if necessary, and fills in the body. Next, the busy flag in the Level 0 header is set and the message is added to the active list. Finally, the message

is placed in the receiving processor's Level 0 queue, and the sending processor returns to whatever it was doing. At the receiving end, during clock interrupts and when the processor is idle, an interrupt handler polls the Level 0 queue for that processor. If there are any new messages, they are removed from the Level 0 queue and added to an internal message queue, which the program itself polls at convenient intervals. This internal message queue serves to isolate the "user level" program (coordinator or slave) from the "interrupt level" message handling, and allows the program to synchronize message processing with its own internal operation. To process a message, the user program removes it from the internal queue and dispatches on the Opcode field to the appropriate handler routine. When the handler is finished, it clears the busy flag in the message and returns. The sending program periodically searches through its list of active messages, reclaiming those that are no longer in use.

On top of the non-blocking message passing mechanism described above, a simple synchronous request/reply scheme was implemented. This feature is used primarily for debugging purposes and to answer queries from the user. For example, the slave processors use this mechanism to obtain the ASCII name of a node from the coordinator when printing debugging information. The *RequestID* field of the message is used to match incoming replies with outstanding requests. All other messages are left in the queue unprocessed until all pending requests have received replies.

### 3.4. History Mechanism

Chapter Two discussed the requirements the history maintenance mechanism must meet. These are summarized below.

- The checkpoint must contain *all* of the information necessary to completely and atomically transform one consistent simulation state to another. There must be no period in which inconsistent results may be given.
- It must be possible to make forward progress under all possible circumstances. This does not imply we must make forward progress after *every*

roll back, but eventually the simulation must converge.

In addition to meeting the above constraints, we would like the history mechanism to be efficient in both time and memory, as these costs represent part of the overhead associated with parallel execution.

### 3.4.1 Simulation State Information

We can take advantage of the nature of the simulation algorithm to minimize the amount of state information that must be checkpointed. As shown in Chapter Two, this information includes the internal state of the circuit, the state of externally applied inputs, and the state of the algorithm itself. The state of the circuit consists of the logic state of each node in the network. The history of externally driven node values comes for free by maintaining the input list throughout the simulation. The state of the simulation algorithm consists of the contents of the event lists and the current simulated time. Since checkpointing and roll back occur only at specified places in the slave program, no other process state (*i.e.*, the stack) need be saved.

All of the state information is kept in a data structure known as the *checkpoint* structure. The list of extant checkpoint structures is kept sorted by simulated time. The data structure contains a time stamp to identify the simulated time the checkpoint was taken, an array of pointers to the saved event lists, and an array of node values. The procedure for filling the checkpoint structure is described below.

1. Allocate a new checkpoint data structure. Mark it with the current simulated time and add it to the end of the checkpoint list.
2. Make a copy of each event in the event wheel and add it to the appropriate list in the checkpoint structure's event array.
3. Visit each node in the network, recording its value in the node array of the checkpoint structure.

For each node in the network, the checkpoint procedure must record its state (*0*, *1*, or *X*) and whether the user has declared it to be an input. Therefore, three bits

of information are needed to completely specify the state of a node. For the sake of simplicity and performance, two nodes are packed into each byte of the node array (it would be more storage efficient but slower to store 5 nodes per 16-bit word). The procedure to checkpoint the state of the network is shown in Figure 3.5.

```
/* Array is the node array of the checkpoint structure */
CkptNetwork(Array)
  char *Array;
  { int Index := 0;
    for each node in the network, n
      { /* Even nodes are put in low order nibble */
        if Index is even
          { Array[Index] := NodeValue(n);
            if n is an input
               Array[Index] := Array[Index] ORed with 0x04;
          }
        /* Odd nodes are put in high order nibble */
        else
          { Array[Index] := Array[Index] ORed with
                NodeValue(n) shifted left by 4 bits;
            if n is an input
               Array[Index] := Array[Index] ORed with 0x40;
            Index++;
          }
      }
  }
```

**Figure 3.5.** Checkpointing the Network State

### 3.4.2 Checkpoint Strategy

In Chapter 2 we discussed a strategy to vary the frequency of checkpointing to achieve both a high density of checkpoints in the vicinity of communication clusters, and a low average overhead when the simulation is well balanced. To this end, we define a *checkpoint cycle* to be the set of checkpoints between any pair of occurrences of resynchronization or roll back.

Figure 3.6 demonstrates the strategy chosen. The checkpoint cycle begins at time $t_0$. The checkpoints are indicated by Xs. If this cycle was initiated by a resynchroniza-

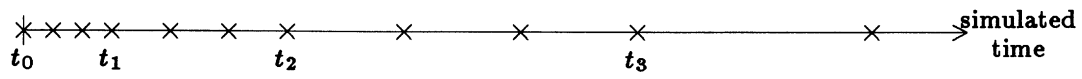


**Figure 3.6.** Checkpoint Distribution

tion, a checkpoint is taken at $t_0$ to guarantee the simulation can be rolled back to its initial state. If the cycle was initiated by a roll back to $t_0$, the checkpoint at $t_0$ is still valid, so no new checkpoint is taken. In either case, the state is then checkpointed at each succeeding time step for the next three steps, ensuring forward progress will be made. At time $t_1$ the period increases to two steps, at $t_2$ the period increases to four steps, and so on. The period increases in this fashion to a maximum period of 1024 time steps. Both the time constant and the final value of the exponential were chosen empirically.

## 3.5. Roll Back Mechanism

The queue of incoming messages is examined at the end of the first phase of the simulation loop. If there are any input change messages pending, they are removed from the queue and processed. For each entry in each message, an input change structure is inserted into the input list at a place specified by the simulated time contained in the message. Let $t_0$ be the simulated time specified in the earliest pending message. If `CurTime`$\leq t_0$, no further action is taken. If `CurTime`$> t_0$, the processor must stop the simulation and roll back. To roll back, the processor walks back through the checkpoint list to find the latest checkpoint taken at a time $t_c \leq t_0$. Each node of the partition is visited and its value restored from the node array of the checkpoint structure. All events currently on the event lists are thrown away, and the event lists in the checkpoint structure are copied into their places. The `NextInput` pointer is moved back through the input change list to point to the next change at time $t_i \geq t_c$. A roll back notification message is sent to the coordinator and to all other partitions dependent upon this one. Finally, all checkpoints taken after $t_c$ are reclaimed for later use (added to a free list). Details of the roll back operation are shown in Figure 3.7. The `RestoreNetwork` routine

```
/* Roll the simulation back to a time before t and restore
 * the state from event checkpoint and node history lists
 */
RollBack(t)
  int t;
  { struct checkpoint *ctmp;
    /* find closest checkpoint to roll back time t */
    ctmp := last element of CkptList;
    while time of ctmp > t
        ctmp := previous element of CkptList;
    CurTime := time of ctmp;
    /* walk the network restoring node values */
    RestoreNetwork(ctmp);
    /* restore event array and overflow list */
    RestoreEvents(ctmp);
    /* back up next input pointer */
    while scheduled time of NextInput ≥ CurTime
        NextInput := previous element of InputList;
    /* Roll back notification to anyone who cares */
    for each partition in dependent list
        send roll back notification;
    /* garbage collect old checkpoints */
    for each checkpoint in CkptList > CurTime
      { remove from CkptList;
        place on FreeCkptList;
      }
  }
```

**Figure 3.7.** Roll Back Procedure

is similar to the CkptNetwork routine discussed earlier.

When processor $P_i$ receives notification that processor $P_j$ rolled back to time $t_0$, $P_i$ must clean up its act to reflect the new knowledge about the state of $P_j$. If $P_i$ has no record of input changes from $P_j$ which are dated more recently than $t_0$, nothing need be done. If $P_i$ has changes from $P_j$ more recent than $t_0$, those changes are spliced out of the input list. If $P_i$ has not processed any of those changes (*i.e.* the earliest change is scheduled for a time > $\texttt{CurTime}_i$), no further action is taken. If, however, $P_i$ has already processed at least one of the changes, the results of those changes must be undone. $P_i$ must therefore roll back to a time preceding the earliest of the invalid

changes. Note that $P_i$ need not be rolled all the way back to $t_0$, but only far enough to undo the effects of false changes from $P_j$. Any new changes from $P_j$ will explicitly force $P_i$ to roll back. This response is shown in more detail in Figure 3.8. The history and roll back mechanisms are presented in Appendix B.

```
/* Respond to Roll Back Notification from processor P at time t *\
HandleNotify(P, t)
  { int earliest;
    struct Input *in;
    /* walk backward from end of InputList to remove inputs from P */
    in := last element of InputList;
    while scheduled time of in > t do
      { if in came from processor P
          { earliest := scheduled time of in;
            remove in from InputList;
          }
        in := previous element of InputList;
      }
    /* Roll back to earliest, if necessary */
    if (CurTime > earliest)
        RollBack(earliest);
  }
```

**Figure 3.8.** Response to Roll Back Notification

## 3.6. Summary

PRSIM is a logic level simulator based upon the RSIM algorithm which takes advantage of the locality of circuit operation to achieve parallelism. Interprocessor synchronization is accomplished through the history maintenance and roll back technique presented in Chapter Two. PRSIM makes few demands upon the underlying parallel architecture. It requires a reliable, order preserving message delivery substrate for communication. There is no need for shared memory, or special hardware for floating point arithmetic or memory management. The current implementation of PRSIM has no automatic partitioning mechanism. The designer must specify the partitioning before simulation.

The kernel of the original RSIM program (excluding user interface) consists of approximately 1430 lines of C code. The simulation slave portion of PRSIM, including message handling, contains approximately 2800 lines of C code, or roughly double the original size. Of the 2800 lines, approximately 450 lines are dedicated to the history maintenance and roll back features, while message handling, file I/O, and debugging account for the rest. There are about 800 lines of code dedicated to the coordinator's administrative functions (excluding user interface), split roughly evenly between file I/O and message management.

Chapter IV

# Results

A preliminary set of experiments were designed and run to determine the performance of the PRSIM implementation. The first set of experiments were designed to measure the overall performance of PRSIM, with special emphasis on the scaling behavior. To completely understand the results of these experiments, extensive performance monitoring facilities were added, and a second set of experiments run. This chapter presents and discusses the results from those two sets of experiments.

## 4.1. Overall Performance

### 4.1.1 Experimental Procedure

To determine the scaling behavior of PRSIM, a set of identical simulations were run with a varying number of processors. The set of simulations is composed of one test circuit and a large number of randomly generated test vectors. The experiments consisted of simulating all of the vectors on each of a number of partitionings of the test circuit.

The number of *essential events* for a given circuit and set of test vectors is defined

to be the number of events processed in a uniprocessor simulation. This set of events is the standard by which multiprocessor simulations are judged. Therefore, the number of essential events processed per second of run time is a measure of the useful (non-redundant) work performed. This is the metric by which the overall performance of the parallel simulator is measured. To obtain these values, it is necessary to count the number of events processed in the one partition experiment, and the amount of time elapsed during the simulation of each vector in each experiment. Elapsed time is measured in units of clock ticks, where one clock tick $\approx$ 16.2mSec.

The scaling behavior is most easily expressed in terms of the effective speedup factor obtained from a given number of processors. The speedup factor for $N$ processors is defined to be:

$$Speedup = \frac{t(N)}{t(1)}$$

where $t(N)$ is the time taken to run a given experiment on $N$ processors. The extra simulation incurred as a result of roll back can be expressed in terms of the *simulation efficiency*, which is defined to be:

$$\eta_s = \frac{No.\ of\ events(1)}{No.\ of\ events(N)}$$

where $No.\ of\ Events(N)$ is the number of events processed in an $N$ partition experiment.

The test circuit is a 64-bit adder, using a dynamic logic CMOS technology. The adder uses a look ahead carry mechanism for groups of four bits, as shown in Figure 4.1. The dynamic logic is clocked by a two phase clock, supplied externally. The carry out signal from each group is rippled into the next most significant group of the adder. Because the dynamic logic in the carry look ahead block is highly connected, the adder will be partitioned along the four bit group boundaries. The only communication between the partitions consists of the carry chain. The adder contains a total of 2688 transistors and 1540 nodes. There are 1328 N-type transistors, 1360 P-type transistors. Each 4-bit slice contains 168 transistors, and 96 nodes.

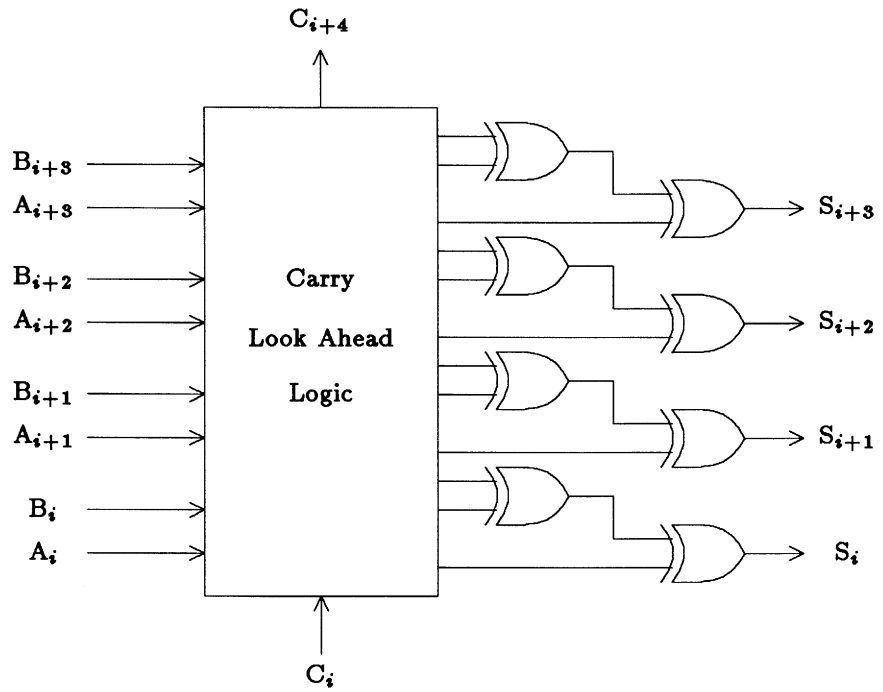


**Figure 4.1.** 4-Bit Slice of Adder Circuit

Experiments were run with the test circuit partitioned into 1, 2, 3, 4, and 6 partitions. The organization of each partition is shown in Figure 4.2. The marks across the top indicate groups of four bits. In each experiment, all of the partitions are of equal length except the six partition case, where the first two partitions contain 8 bits each, while the rest contain 12 bits. Random test vectors of varying length were used. The lengths ranged from 2 to 24 clock cycles, with four sets of vectors in each length.

### 4.1.2 Results

A summary of the raw performance data is shown in Table 4.3. The complete results are presented in Appendix C. Table 4.3 shows the average performance of PRSIM in essential events per second as a function of both the length of the test vector and the number of processors. There are a number of discrepancies from what might be considered ideal behavior. The first is the decline in raw performance of the one

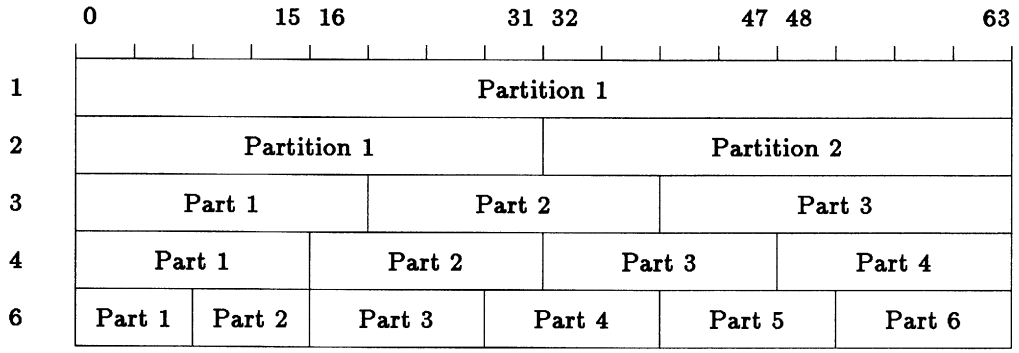


**Figure 4.2.** Adder Partitioning

partition experiment as the length of the test vector increases. This is attributed to the cost of reclaiming the checkpoint data structures upon resynchronization. Since each checkpoint contains an arbitrary number of events, it necessary to walk the length of the checkpoint list when reclaiming, incurring a cost proportional to the length of the list.

| | Number of Processors | | | | |
|---|---|---|---|---|---|
| Length | 1 | 2 | 3 | 4 | 6 |
| 2 | 46.45 | 81.09 | 107.62 | 134.50 | 151.52 |
| 4 | 44.55 | 80.60 | 108.10 | 129.36 | 166.41 |
| 6 | 42.95 | 76.25 | 98.99 | 136.56 | 170.93 |
| 8 | 41.39 | 76.87 | 100.72 | 126.25 | 155.75 |
| 12 | 40.62 | 78.32 | 105.45 | 126.42 | 159.19 |
| 16 | 38.86 | 75.65 | 96.08 | 126.49 | 152.97 |
| 24 | 37.66 | 74.94 | 94.24 | NA | 145.48 |

**Table 4.3.** Raw Performance Results in Events/Second

| N | $\eta_s$ | Speedup |
|---|---|---|
| 1 | 1.000 | 1.00 |
| 2 | 0.991 | 1.86 |
| 3 | 0.967 | 2.43 |
| 4 | 0.937 | 3.11 |
| 6 | 0.951 | 3.77 |

**Table 4.4.** Simulation Efficiency and Speedup Factor

Table 4.4 shows the average simulation efficiency and the speedup factor as a

function of the number of processors. Table 4.4 demonstrates that the simulation efficiency is relatively unaffected by the number of partitions. This indicates a both high degree of decoupling between the partitions, with a corresponding low occurance of roll back, and a even balance in simulation load, which is consistent with the partitioning chosen.

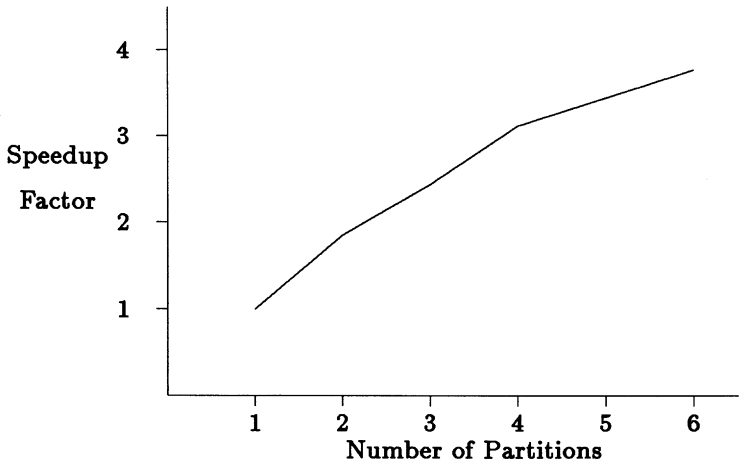


**Figure 4.5.** Speedup Factor Versus Number of Partitions

The speedup factor results are somewhat more interesting. Figure 4.5 presents a plot of the speedup as a function of the number of partitions. With six processors, a performance improvement of only 3.77 is achieved. The performance increases less than linearly with the number of processors. Clearly, the small decrease in simulation efficiency is not the dominant factor. To understand this phenomenon, more detailed information is required.

## 4.2. Profiling Results

To understand the performance behavior of PRSIM, it is necessary to build a detailed model of the costs associated with the various functions. In particular, we need to know the following information:

1. Impact of the partitioning – How well balanced is the simulation load? How much interprocessor communication is there?

2. Synchronization costs – How much time is spent maintaining the checkpoint lists? How expensive is the roll back operation?

3. Communication costs – How expensive is message handling? How much of that cost is associated with the low level implementation?

To obtain this information, a statistical profiling scheme similar to that of Version 7 UNIX† was implemented for the simulation slave program.

### 4.2.1 Experimental Procedure

The profiling scheme collects the number of calls to every subroutine in each simulation slave and total amount of time spent in each subroutine. This information is sufficient to determine the percentage of the total time that is spent in each subroutine, and the average length of time spent per subroutine call.

When the program starts up, a table to contain the subroutine call information is built. Each line of the table contains a pointer to the entry point of one subroutine, and the count of the number of times that routine has been called. Each subroutine contains a pointer to the corresponding table entry. The compiler automatically inserts code at the beginning of every subroutine to manipulate the count table. When the routine is first called, it is linked into the table. On each succeeding call, the corresponding counter in the table is incremented. When the program exits, the table is written into a file to be interpreted later.

A statistical averaging technique is used to determine the amount of time spent in each subroutine of the program. A table of program counter ranges is maintained in which each entry represents the number of times the sampled program counter lay within a given 8 byte range. At every clock interrupt (once every 16mSec.), the program counter is sampled, the value shifted right 3 bits, and used as an index into the array. The indexed table entry is then incremented. When the program exits, the table is written into a file to be interpreted later. By taking a sufficiently large number of

† UNIX is a trademark of Bell Laboratories.

samples, we can obtain a fairly accurate profile of the amount of time spent in each subroutine.

Profiling data was gathered for the six partition experiment described above. Five sets of test vectors, each of length 16, were run. To provide a sufficiently large sample, each vector was simulated ten times. Therefore, the sample consists of 800 simulated clock cycles of 200nSec. each, or 160$\mu$Sec. of simulated time. Each vector generates roughly 18,000 essential events, for a total of approximately 850,000 events in the sample.

Profiling is enabled by the coordinator immediately before the input vectors are established, and disabled immediately after each resynchronization. Therefore, the profiling data does not include time spent in the user interface.

### 4.2.2 Results

The complete results of the profiling experiment appear in Appendix D. Table 4.6 summarizes the percentage of idle time recorded by each processor (time spent in the routine `step`). The idle time is the sum of the time elapsed between reaching the specified stop time and the subsequent resynchronization or roll back. The high idle times of partitions #1 and #2 are the result of the relative partition sizes: partitions #1 and #2 contain 8 bits each, while the rest contain 12 bits. The decrease in the idle times from partition #3 to #6 follows the communication through the carry chain: the further down the chain, the longer it takes to settle.

The speedup results reported earlier can now be explained. The expected speedup for $N$ processors can be expressed as:

$$Speedup = N\eta_s\eta_p$$

where $\eta_p$ is the processor utilization factor. For the six partition experiment, we obtain an expected speedup of 4.03. The non-linearity of the curve can be explained by $\eta_p$ decreasing as the number of partitions is increased.

| Partition | % Idle |
|---|---|
| 1 | 50.04 |
| 2 | 50.27 |
| 3 | 28.99 |
| 4 | 21.67 |
| 5 | 16.13 |
| 6 | 3.80 |
| Total | 29.14 |

**Table 4.6.** Idle Time per Partition

Table 4.7 shows a break down of where the active (non-idle) time was spent by each partition. The figures are percentages of the total active time of each partition. The data is divided into three categories of activity as follows:

*Simulation:* the time spent in the RSIM simulation algorithm itself. This is subdivided as follows:

*Arithmetic:* the time spent in the scaled fixed point arithmetic routines.

*Other:* all other aspects of the RSIM algorithm.

*History:* the time which may be attributed to the roll back synchronization scheme. This is subdivided as follows:

*Checkpoint:* the time spent creating and maintaining the state checkpoints.

*Roll Back:* the time spent restoring the state upon roll back.

*Communication:* the time associated with interprocessor communication. This is subdivided as follows:

*System Level:* the time spent polling and manipulating the interrupt level message queues.

*User Level:* the time spent constructing and handling messages at the user level.

Table 4.7 shows the amount of time spent in overhead is relatively small; nearly 90% of the active time is spent in the simulation algorithm, most of that in the fixed point routines. The overhead time is dominated by the communication, and not by the history mechanism. Only in partition # 2, which had a relatively high incidence of roll

| | Partition Number | | | | | | |
|---|---|---|---|---|---|---|---|
| Function | 1 | 2 | 3 | 4 | 5 | 6 | Total |
| Simulation | | | | | | | |
| Arithmetic | 60.43 | 58.17 | 63.63 | 64.18 | 64.05 | 65.08 | 63.07 |
| Other | 25.94 | 25.07 | 25.16 | 25.19 | 25.71 | 25.17 | 25.35 |
| Total | 86.37 | 83.24 | 88.79 | 89.37 | 89.76 | 90.25 | 88.42 |
| History | | | | | | | |
| Checkpoint | 2.93 | 5.68 | 2.47 | 2.64 | 2.54 | 2.70 | 3.01 |
| Roll Back | 0.00 | 0.30 | 0.00 | 0.05 | 0.05 | 0.07 | 0.07 |
| Total | 2.93 | 5.98 | 2.47 | 2.69 | 2.59 | 2.77 | 3.08 |
| Communication | | | | | | | |
| System Level | 3.56 | 3.57 | 2.54 | 2.31 | 2.22 | 1.98 | 2.57 |
| User Level | 7.14 | 7.21 | 6.20 | 5.63 | 5.43 | 5.00 | 5.93 |
| Total | 10.70 | 10.78 | 8.74 | 7.94 | 7.65 | 6.98 | 8.50 |

**Table 4.7.** Breakdown of Time Spent by Function

back, is the checkpointing overhead non-negligable.

## 4.3. Discussion

There are two important conclusions that can be reached from the results reported in this chapter. First, the circuit partitioning has a significant impact on the scaling performance of the simulator. The dominant effect, at least in the small test case reported here, is not the overhead associated with communication or synchronization, but is the dynamic load balance. Even though the test circuit was statically well partitioned, the dynamic behavior resulted in only about 70% processor utilization with six partitions. Decreasing processor utilization resulted in "diminishing returns" in the speedup factor, as shown in Figure 4.5.

The second conclusion is that the results reported are inconclusive. Because the active time was so completely dominated by the simulation load, it is difficult to build any detailed models of the overhead costs associated with the history and roll back mechanisms. The test circuit was too small and too regular to exhibit much interesting behavior. Somewhat better results could perhaps have been achieved by running much longer test vectors. Unfortunately, the current implementation of PRSIM is severely memory bound. If automatic resynchronization were employed to limit the storage

required for history maintenance, larger experiments could be performed.

Chapter V

# Conclusion

## 5.1. Summary

Integrated circuit technology has been advancing at a phenomenal rate over the last several years, and promises to continue to do so for the foreseeable future. If circuit design is to keep pace with fabrication technology, radically new approaches to computer-aided design will be necessary. This thesis has explored the problems of capacity limitation in existing simulation tools, and has sought to develop a new approach to building fast, scalable circuit simulators.

We began by examining the locality inherent in digital circuit operation. Digital circuit elements operate on local information, producing local results. It was observed that there exists a class of simulation algorithms which exhibit a similar locality property. Therefore, we set out to develop a framework for circuit simulation which could take advantage of this locality to achieve a high degree of parallelism. The scheme we developed involved mapping the circuit to be simulated onto the topology of the target multiprocessor to take advantage of the natural structure of the circuit. We explored

the problems associated with the precedence constraints imposed by the partitioning. We discovered that many of these problems could be avoided by inserting buffers between the partitions, effectively decoupling them. This lead to the development of a synchronization mechanism based upon history maintenance and roll back. By periodically saving the state of the simulation, each partition can be allowed to simulate asynchronously with respect to the others, rolling back the simulation if necessary to correct for input changes. This solution was demonstrated to be sufficient to guarantee convergence in the presence of feedback. We then discussed the importance of the strategy used to partition the circuit, and argued that static graph partitioning techniques may not be adequate. Finally, we quickly reviewed some related research, with emphasis on the relationship to the work report in this thesis.

To determine the merit of the ideas presented in Chapter Two, a circuit simulator, PRSIM, was designed and built. Chapter Three discussed the details of that implementation. The chapter began with an overview of the RSIM simulator and the Concert multiprocessor on which PRSIM is based. RSIM was chosen as the vehicle for this implementation because it is an event driven simulator which exhibits the locality properties discussed in Chapter Two. PRSIM is organized into three components: the prepass phase, which is responsible for partitioning the circuit; the coordinator, which is responsible for attending to the administrative functions, such as file I/O and interfacing to the user; and the simulation slave, which performs the actual work of the RSIM algorithm. We discussed the organization of the simulation control loop, which is decomposed into two phases: an event processing phase, and an input processing phase. This two phase organization, together with the variable checkpointing strategy, is sufficient to guarantee convergence according to the argument presented in Chapter Two. All interprocessor communication is implemented by a simple, non-blocking message passing mechanism, built on top of the Concert Level 0 message passing protocol. Some optimizations were made in light of the fact that Concert is a tightly coupled multiprocessor system, but the essential mechanism does not rely upon shared memory.

Finally, the history maintenance and roll back algorithms were presented in detail.

A preliminary set of experiments were run to determine the scaling behavior of PRSIM. The experiments were organized into two sets. The first set was designed to measure the overall performance of PRSIM, while the second set was designed to obtain detailed information about the internal behavior of PRSIM. From the first set, we learned that the performance increased by nearly a factor of 2 in going from one to two partitions, but that beyond two there was a "diminishing returns" phenomena. From the profiling experiments of the second set, we discovered the cause of this behavior was the processor utilization decreased as the circuit was partitioned into finer and finer pieces. The profiling experiments also revealed that less than 12% of the active processing time was spent in overhead associated with parallel execution. Although this result was somewhat encouraging, it made it nearly impossible to develop models of the overhead costs and scaling behavior. The conclusion derived from these results is that the partitioning strategy is very important, and requires further research.

## 5.2. Directions for Future Research

The results reported in this thesis suggest several avenues for future research. One of the most serious problems encountered was the bound on the length of the simulation which resulted from the memory requirements of checkpointing. This suggests the need for automatic resynchronization: reclaiming old state once it can be guaranteed that no partition can be forced to roll back beyond a certain point. This will require additional communication to coordinate the checkpointing, but is probably cost effective in the long run.

The checkpointing strategy that was implemented was based upon empirical results with arbitrarily chosen parameters. One direction for future work is to develop formal statistical models for the communication behavior of digital circuits. These models could then be used to optimize the checkpointing strategy for a particular circuit, either statically at partition time, or dynamically based upon the communication patterns observed.

Perhaps the most important issue raised is the problem of effective network partitioning. It would be interesting to explore the limits of static partitioning algorithms. Ultimately, however, it will probably be necessary to turn to some form of dynamic partitioning. Two quantities determine the performance of a given partitioning: the amount of useful simulation work accomplished by each partition, and the amount of communication among the partitions. A dynamic partitioning strategy should try to balance the first quantity, while minimizing the second. We can view the level of activity in a partition as a "temperature". As the activity (simulation work and communication) increases, the temperature rises. The goal of dynamic partitioning is to achieve a low, uniform temperature across the multiprocessor. Periodically, the temperature of each partition should be sampled, and atomic units from hotter partitions moved into adjacent, cooler partitions, following the temperature gradient. If the fluctuations in temperature have a very short time constant (on the order of a single clock cycle), it may only be necessary to repartition once or twice near the beginning of a simulation.

The framework that we have described does not rely upon the memory architecture of any particular multiprocessor. It is intriguing to consider the possibility of a simulation spread among a loosely coupled collection of machines. For example, it should be possible to build a simulator which locates idle machines on a local area network, and dispatches pieces of the simulation load to them. To determine the viability of this idea, we need a better understanding of the sensitivity of our approach to message latency. A series of experiments can be performed with the current PRSIM implementation in which the message delivery latency is varied by the sending processor.

A great deal of the active run time of PRSIM was spent in the fixed point arithmetic package. Although not directly related to the field of parallel simulation, this problem suggested the construction of an assigned delay simulator. The prepass phase of such a simulator would construct a table of transition delays for each node in the circuit using the RSIM (or any other) model. Having thus precomputed the delays for every node in the circuit, at run time the simulator need only perform a table look up to schedule an

event.

## 5.3. Conclusion

We have presented an approach to parallel simulation which is based upon the inherent parallelism of circuit operation. The initial implementation of PRSIM demonstrates that history maintenance and roll back is a viable solution to interprocessor synchronization in this context. Much work remains to be done, however, to determine whether this approach can indeed be scaled to an arbitrary number of processors.

Appendix A

# PRSIM Messages

The following is a list of the message types used by PRSIM. The entry for each message contains the name of the message, the purpose of the message, and the information contained in the body. The messages are divided into three groups: coordination of the simulation; file I/O with the host computer; and support for the user interface.

The following group of messages support the coordination of the simulation activity.

## LOAD-NETWORK

The LOAD-NETWORK message is sent from the coordinator to each simulation slave upon initialization. This message contains the number of partitions in the simulation, the partition ID for the receiving processor, the table to map partition numbers to processor numbers, and the name of the partition file on the host computer.

## LOAD-NETWORK REPLY

The LOAD-NETWORK REPLY message is sent by each slave to the coordinator upon the completion of the network initialization. The body is empty.

Appendix A

# PRSIM Messages

The following is a list of the message types used by PRSIM. The entry for each message contains the name of the message, the purpose of the message, and the information contained in the body. The messages are divided into three groups: coordination of the simulation; file I/O with the host computer; and support for the user interface.

The following group of messages support the coordination of the simulation activity.

**LOAD-NETWORK**

The LOAD-NETWORK message is sent from the coordinator to each simulation slave upon initialization. This message contains the number of partitions in the simulation, the partition ID for the receiving processor, the table to map partition numbers to processor numbers, and the name of the partition file on the host computer.

**LOAD-NETWORK REPLY**

The LOAD-NETWORK REPLY message is sent by each slave to the coordinator upon the completion of the network initialization. The body is empty.

**SETNODE**

The SETNODE message is sent to a slave to inform it of external node changes. The body contains the simulated time the change took place, and a list of (node ID, new value) pairs.

**STEP**

The STEP message is sent from the coordinator to all slave processors to initiate a simulation step. The body contains the simulated time the step is to terminate.

**SETTLED**

The SETTLED message is sent from a slave to the coordinator to notify it that the slave has reached the specified termination time. The body contains the partition number of the sender.

**ROLLBACK**

The ROLLBACK message is sent from a slave to the coordinator and all dependent partitions to notify them that the slave has rolled back its simulation. The body of this message contains partition number of the sender, and the simulated time the partition rolled back to.

**RESYNC**

The RESYNC message is sent from the coordinator to all slave processors to inform them the simulation has settled. The slave processors use this information to reclaim the storage in the checkpoint and input lists. The body of this message is empty.

The following group of messages implements remote file access.

**FOPEN**

The FOPEN message is a request from a slave to the coordinator to open the named file on the host computer. Only one open file is allowed at any one

time. The body contains the host file name and the access mode (*e.g.*, read or write).

### FOPEN REPLY

The FOPEN REPLY message informs the slave the requested file is open and ready for use. The body contains a single integer reflecting the result of the open operation: a 0 indicates a successful open, a $-1$ indicates an error.

### FREAD

The FREAD message is a request from a slave to the coordinator to read a block of data from the open file. The body contains the number of items to be read, and the size of each item. A maximum of 1024 bytes may be requested.

### FREAD REPLY

The FREAD REPLY message contains the data requested by a FREAD message. The body contains the number of items read and the data read.

### FWRITE

The FWRITE message is a request from a slave to the coordinator to write a block of data to the open file. The body contains the number of items to be written, the size of each item, and the data to be written.

### FWRITE REPLY

The FWRITE REPLY message reports the result of a FWRITE message. The body contains an integer error value which is 0 if the write was successful, $-1$ if the write failed.

### FCLOSE

The FCLOSE message is a request from a slave to the coordinator to close the opened file. No reply is necessary.

The following group of messages support the user interface.

### PRINTF

The PRINTF message is sent by a slave processor to the coordinator to print an arbitrary string on the user's console. The body contains the null terminated ASCII string to be printed. The coordinator prefixes the partition ID of the slave to the string before printing.

### GETNODE

The GETNODE message is a request by the coordinator to obtain the current value of a given node from a slave. The body contains the global ID of the node.

### GETNODE REPLY

The GETNODE REPLY message is the reply from a slave to the coordinator to a GETNODE request. The body contains the value of the requested node.

### NODE-INFO

The NODE-INFO message is a request by the coordinator to obtain connectivity information about a node within the network. This message is originally sent to the slave responsible for driving the node. This slave prints its relevant information for the user (via PRINTF messages), and then forwards the NODE-INFO message to any adjacent partitions. Each adjacent partition sends its information directly back to the coordinator in the form of PRINTF messages, and then replies to the forwarding slave. When all adjacent partitions have replied, the forwarding slave replies to the coordinator. The body of the NODE-INFO message contains the global ID of the requested node and the type of information requested.

### NODE-INFO REPLY

The NODE-INFO REPLY message is sent by a slave partition to the processor

which requested NODE-INFO, after all of the information has been printed. The body contains the partition ID.

### TRACE-NODE

The TRACE-NODE message is sent by the coordinator to enable activity tracing for a particular node. The body contains the global ID of the node to be traced. The receiving partition sets a flag in the specified node to enable tracing. Whenever a traced node changes value, a notice is printed on the user's console.

### UNTRACE-NODE

The UNTRACE-NODE message is sent by the coordinator to cancel activity tracing for a particular node. The body contains the global ID of the node.

### GETNAME

The GETNAME message is sent by a slave processor to the coordinator to request the ASCII name of a given node. The body contains the global ID of the node. This message is used when printing node information on the user's console.

### GETNAME REPLY

The GETNAME REPLY message is the coordinator's reply to the GETNAME message. The body contains a null terminated ASCII string representing the name of the requested node.

### DEBUG-LEVEL

The DEBUG-LEVEL message is sent from the coordinator to all slave processors to set the debug level. The value in the body determines the type and quantity of debugging information to display. There is no reply.

**ENABLE-PROFILE**

The ENABLE-PROFILE message is sent from the coordinator to all slave processors to enable the performance monitoring software. The body is empty, and there is no reply.

**DISABLE-PROFILE**

The DISABLE-PROFILE message is sent from the coordinator to all slave processors to disable the performance monitoring software. The body is empty, and there is no reply.

Appendix B

# History Implementation

```
/* This file contains the implementation of the history
 * maintenance and roll back mechanisms of PRSIM.
 *
 * A few globally defined structures are reproduced below.
 */

/* Useful data structures */

struct Event {                    /* the structure of an event */
  evptr flink,blink;              /* doubly-linked event list */
  nptr enode;                     /* node this event is all about */
  long ntime;                     /* time, in DELTAs, of this event */
  char eval;                      /* new value */
  char type;                      /* type of event */
};

struct Checkpt                    /* the structure of a checkpoint */
  { ckptr flink, blink;           /* double linked list checkpoint list */
    long ctime;                   /* time checkpoint was taken */
    int ev_index;                 /* index into event array */
    struct Event *event[TSIZE]; /* copy of event array */
    struct Event *overflow;       /* copy of overflow event list */
    char *svect;                  /* pointer to node state table */
};

struct Input {                    /* linked list of inputs */
  iptr next;          /* next element of list */
```

```
  nptr inode;          /* pointer to this input node */
};

/* For convenience, pointers are abbreviated as follows */
typedef struct Event *iptr;      /* event pointer */
typedef struct Evckpt *ckptr;    /* checkpoint pointer */
typedef struct Input *iptr;      /* input pointer */

/* Routine to checkpoint the state of the simulation
 * Note that the checkpointed event array & overflow list are stored more
 * compactly than the originals.
 */

checkpoint()
  { register ckptr ctmp;
    register evptr etmp, ev, ev_base;
    register int i, j;
    char *ptr;

  /* get a ckpt structure from free list, allocating more if neccessary */
    if ((ctmp = ck_free) == NULL)
      { ctmp = (ckptr)al_bytes(10 * sizeof(struct Evckpt));
        ptr = (char *)al_bytes(nums10);
        for (i = 10; --i > 0; ctmp++)
          { ctmp->flink = ck_free;
            ck_free = ctmp;
            ctmp->svect = ptr;
            ptr += nums;
          }
        ctmp->svect = ptr;
      }
    else ck_free = ctmp->flink;

  /* add new ckpt struct to list of checkpoints */
    ctmp->flink = &ck_list;
    ctmp->blink = ck_list.blink;
    ck_list.blink->flink = ctmp;
    ck_list.blink = ctmp;

  /* copy event array into ckpt struct */
    for (i = 0; i < TSIZE; i++)          /* loop over lists in array */
      { ev_base = &ev_array[i];
        ev = ev_base;
        ctmp->event[i] = NULL;
        if (ev->flink == ev)             /* if it's empty, do nothing */
            continue;
        while ((ev = ev->flink) != ev_base)  /* loop over each event in list */

        /* allocate event struct */
          { if ((etmp = evfree) == NULL)
```

```
            { etmp = (evptr)al_bytes(10 * sizeof(struct Event));
              for (j = 10; --j > 0; etmp++)
                { etmp->flink = evfree;
                  evfree = etmp;
                }
            }
          else evfree = etmp->flink;

        /* copy contents of old (ev) to new event */
          etmp->enode = ev->enode;
          etmp->ntime = ev->ntime;
          etmp->eval  = ev->eval;
          etmp->type  = ev->type;
        /* add new event to checkpoint event array */
          if (ctmp->event[i] == NULL)
              etmp->flink = etmp->blink = ctmp->event[i] = etmp;
          else
            { etmp->flink = ctmp->event[i];
              etmp->blink = ctmp->event[i]->blink;
              ctmp->event[i]->blink->flink = etmp;
              ctmp->event[i]->blink = etmp;
            }
        }
      }

/* copy overflow array into ckpt struct */
    ev = &overflow;
    ctmp->overflow = NULL;
    if (ev->flink != ev)
        while ((ev = ev->flink) != &overflow)
    /* allocate event structure */
          { if ((etmp = evfree) == NULL)
              { etmp = (evptr)al_bytes(10 * sizeof(struct Event));
                for (j = 10; --j > 0; etmp++)
                  { etmp->flink = evfree;
                    evfree = etmp;
                  }
              }
            else evfree = etmp->flink;
    /* copy contents of old (ev) to new event */
            etmp->enode = ev->enode;
            etmp->ntime = ev->ntime;
            etmp->eval  = ev->eval;
            etmp->type  = ev->type;
    /* add new event to checkpoint event array */
            if (ctmp->overflow == NULL)
                etmp->flink = etmp->blink = ctmp->overflow = etmp;
            else
              { etmp->flink = ctmp->overflow;
```

```
                etmp->blink = ctmp->overflow->blink;
                ctmp->overflow->blink->flink = etmp;
                ctmp->overflow->blink = etmp;
              }
          }

/* fill out rest of checkpoint struct */
    ctmp->ctime = cur_delta;           /* time stamp of checkpoint */
    ctmp->ev_index = ev_index;        /* place in event array */
    checkpt_nodes(ctmp);                /* go get node values, too */
    last_ck = cur_delta;               /* remember that we checkpointed */
}

/* Routine to checkpoint the state of the nodes.
 * Walks the network, copying each node value & the state of the
 * INPUT flag into ctmp svect array, two nodes per byte.
 * Argument is a pointer to the checkpoint structure.
 */

checkpt_nodes(ctmp)
  register ckptr ctmp;
  { register nptr n;
    register int i, vindex = 0;
    register char nib = 0, curbyte;
    for (i = 0, vindex = 0; i < HASHSIZE; i++)
        for (n = hash[i]; n; n = n->hnext)
          { if (nib == 0)          /* even nodes in low nibble */
              { nib++;
                curbyte = n->npot;
                if (n->nflags & INPUT) curbyte |= 0x04;
              }
            else                   /* odd nodes in high nibble */
              { nib = 0;
                curbyte |= (n->npot << 4);
                if (n->nflags & INPUT) curbyte |= 0x40;
                ctmp->svect[vindex] = curbyte;
                vindex++;
              }
          }
    if (nib) ctmp->svect[vindex] = curbyte;
}

/* Routine to restore the state of the nodes from a checkpoint.
 * Walks the network, copying each node value & the state of the
 * INPUT flag from ctmp svect array.
 * Argument is a pointer to the checkpoint structure.
 */

restore_nodes(ctmp)
  register ckptr ctmp;
```

```
  { register nptr n;
    register int i, vindex = 0;
    register char nib = 0, curbyte;
    for (i = 0, vindex = 0; i < HASHSIZE; i++)
        for (n = hash[i]; n; n = n->hnext)
          { curbyte = ctmp->svect[vindex];
            if (nib)
              { nib = 0;
                n->ev1 = n->ev2 = NULL;
                n->npot = ((curbyte >> 4) & 0x03);
                if (curbyte & 0x40) n->nflags |= INPUT;
                else n->nflags &=  INPUT;
                vindex++;
              }
            else
              { nib++;
                n->ev1 = n->ev2 = NULL;
                n->npot = (curbyte & 0x03);
                if (curbyte & 0x04) n->nflags |= INPUT;
                else n->nflags &=  INPUT;
              }
          }
}

/* Roll the simulation back to a time before t and restore the state
 * from event checkpoint and node history lists
 */

roll_back(t)
  register long t;
  { register ckptr ctmp;
    register int i, j;
    register evptr ev, etmp, ev_base;
    ckptr nctmp;
    int nevents = 0;
    int oevents = 0;

  /* find closest checkpoint to the roll-back time */
    ctmp = ck_list.blink;
    while (ctmp->ctime > t)
        if (ctmp->blink == &ck_list)
          { error("; roll_back: can't go back to %d",t);
            return 0;
          }
        else ctmp = ctmp->blink;

  /* tell everyone who cares that we're rollin' back */
    rollback_notify(ctmp->ctime);
  /* walk the network restoring node values */
    restore_nodes (ctmp);
```

```
/* restore event array & overflow list, simulated time */
  for (i = 0; i < TSIZE; i++)
    /* free up old current events */
    { ev_base = &ev_array[i];
      if (ev_base->flink != ev_base)
        { ev_base->blink->flink = evfree;
          evfree = ev_base->flink;
          ev_base->flink = ev_base->blink = ev_base;
        }
      /* make a copy of this event list, if there is one */
      if (ctmp->event[i] != NULL)
        { ev = ctmp->event[i];
          do
            /* allocate event struct */
            { if ((etmp = evfree) == NULL)
                { etmp = (evptr)al_bytes(10 * sizeof(struct Event));
                  for (j = 0; --j > 0; etmp++)
                    { etmp->flink = evfree;
                      evfree = etmp;
                    }
                }
              else evfree = etmp->flink;
            /* Copy event data into new event */
              etmp->enode = ev->enode;
              etmp->ntime = ev->ntime;
              etmp->eval = ev->eval;
              etmp->type  = ev->type;
              etmp->flink = ev_base;
              etmp->blink = ev_base->blink;
              ev_base->blink->flink = etmp;
              ev_base->blink = etmp;
            /* link nodes to events */
              if (ev->type == 0)
                  etmp->enode->ev1 = etmp;
              else if (ev->type == 1)
                  etmp->enode->ev2 = etmp;
            }
          while ((ev = ev->flink) != ctmp->event[i]);
        }
    }

/* restore pointer into event array */
  ev_index = ctmp->ev_index;
/* free up current overflow events */
  if (overflow.flink != &overflow)
    { overflow.blink->flink = evfree;
      evfree = overflow.flink;
      overflow.flink = overflow.blink = &overflow;
    }
```

```
/* make a copy of this event list, if there is one */
  if (ctmp->overflow != NULL)
    { ev = ctmp->overflow;
      do
        /* allocate event struct */
        { if ((etmp = evfree) == NULL)
            { etmp = (evptr)al_bytes(10 * sizeof(struct Event));
              for (j = 0; --j > 0; etmp++)
                { etmp->flink = evfree;
                  evfree = etmp;
                }
            }
          else evfree = etmp->flink;
        /* Copy event data into new event */
          etmp->enode = ev->enode;
          etmp->ntime = ev->ntime;
          etmp->eval = ev->eval;
          etmp->type  = ev->type;
          etmp->flink = &overflow;
          etmp->blink = overflow.blink;
          overflow.blink->flink = etmp;
          overflow.blink = etmp;
        /* link nodes to events */
          if (ev->type == 0)
              etmp->enode->ev1 = etmp;
          else if (ev->type == 1)
              etmp->enode->ev2 = etmp;
        }
      while ((ev = ev->flink) != ctmp->overflow);
    }

/* restore current simulated time, and remember there's a
 * good checkpoint here
 */
  cur_delta = ctmp->ctime;
  last_ck = cur_delta;
/* back up input list */
  while ((cur_input->ntime >= cur_delta) && (cur_input != &inlist))
      cur_input = cur_input->blink;

/* garbage collect old checkpoints */
  if (ctmp->flink == &ck_list)
      return;                          /* nothing to collect */
  nctmp = ctmp->flink;                 /* remember next struct in list */
  ctmp->flink = &ck_list;              /* make last struct point to end */
  ck_list.blink->flink = ck_free;      /* old end points to free list */
  ck_list.blink = ctmp;                /* ... and end point to it */
  ctmp = nctmp;
  while (ctmp != ck_free)              /* now collect events inside */
```

```
      { for (i = 0; i < TSIZE; i++)
          { if ((ev = ctmp->event[i]) == NULL) continue;
            ev->blink->flink = evfree;
            evfree = ev;
            ctmp->event[i] = NULL;
          }
        if ((ev = ctmp->overflow) != NULL)
          { ev->blink->flink = evfree;
            evfree = ev;
            ctmp->overflow = NULL;
          }
        ctmp = ctmp->flink;
      }
    ck_free = nctmp;
}

/* Clean up and dispose of ancient history properly
 * We walk the checkpoint list, reclaiming all events inside, and
 * then reclaim the checkpoint list itself.
 * We then move all input changes en masse to the free list.
 * Finally, we take a new checkpoint, just for fun.
 */

cleanup_hist()
  { register ckptr ctmp, nctmp;
    register evptr etmp, ev;
    register int i;

  /* free up all checkpoint structures
   * for each checkpoint, we must first free up all event
   * structures
   */
    ctmp = ck_list.flink;
    while (ctmp != &ck_list)
      { for (i = 0; i < TSIZE; i++)
          { if ((etmp = ctmp->event[i]) == NULL) continue;
            ev = etmp;
            etmp->blink->flink = evfree;
            evfree = etmp;
            ctmp->event[i] = NULL;
          }
        if ((etmp = ctmp->overflow) != NULL)
          {
            ev = etmp;
            etmp->blink->flink = evfree;
            evfree = etmp;
            ctmp->overflow = NULL;
          }
        nctmp = ctmp->flink;
```

```
        ctmp->flink = ck_free;
        ck_free = ctmp;
        ctmp = nctmp;
      }
    ck_list.flink = ck_list.blink = &ck_list;
    last_ck = 0;

  /* flush old input changes (inputs on in_list before current time) */
    ev = &inlist;
    cur_input->blink->flink = evfree;
    cur_input->blink = &inlist;
    evfree = inlist.flink;
    inlist.flink = cur_input;
  /* so's we can roll back to here if need be */
    checkpoint();
}
```

Appendix C

# Raw Performance Data

The following table contains the raw performance data from the experiments described in Section 4.2. The first column contains the name of the test vector, the first component of the name indicates the vector length. The second column contains the number of effective events generated for that vector. The remaining five columns contain the number of clock ticks (16.2mSec/tick) per vector for each experiment.

| Vector | # Events | 1 | 2 | 3 | 4 | 6 |
|---|---|---|---|---|---|---|
| 2A | 1729 | 2242 | 1481 | 950 | 715 | 627 |
| 2B | 1807 | 2323 | 1345 | 1110 | 844 | 885 |
| 2C | 1661 | 2105 | 1112 | 856 | 717 | 571 |
| 2D | 2022 | 2662 | 1443 | 1132 | 967 | 845 |
| 4A | 3678 | 4979 | 2845 | 2114 | 1733 | 1339 |
| 4B | 4000 | 5395 | 3093 | 2328 | 1875 | 1506 |
| 4C | 3977 | 5233 | 2939 | 1941 | 1824 | 1347 |
| 4D | 4045 | 5541 | 2820 | 2380 | 1849 | 1477 |
| 6A | 6714 | 9425 | 5356 | 4279 | 3036 | 2380 |
| 6B | 4510 | 6345 | 3640 | 2685 | 1997 | 1661 |
| 6C | 7196 | 9761 | 5464 | 4162 | 3087 | 2300 |
| 6D | 6573 | 9347 | 5160 | 4054 | 2854 | 2397 |
| 8A | 8414 | 11714 | 6291 | 4793 | 3764 | 3208 |
| 8B | 8535 | 12339 | 6481 | 5052 | 4011 | 3000 |
| 8C | 8140 | 11706 | 6232 | 5280 | 4335 | 3369 |
| 8D | 7817 | 11652 | 6718 | 4524 | 3593 | 3128 |
| 10A | NA | 16925 | 9194 | 7241 | 5037 | 4038 |
| 10B | NA | 16182 | 8435 | 6913 | 4879 | 4361 |
| 10C | NA | 12343 | 6574 | 5184 | 4185 | 3360 |
| 10D | NA | 13287 | 7371 | 4927 | 4776 | 3701 |
| 12A | 13981 | 20453 | 10168 | 7936 | 6907 | 5464 |
| 12B | 14123 | 20285 | 10551 | 7784 | 6383 | 5692 |
| 12C | 10636 | 15928 | 8934 | 6158 | 5484 | 3471 |
| 12D | 11438 | 17328 | 8684 | 6623 | 5080 | 4560 |
| 14A | NA | 21592 | 12090 | 9356 | 7199 | 5327 |
| 14B | NA | 18348 | 9464 | 7023 | 5369 | 4795 |
| 14C | NA | 22572 | 12239 | 8193 | 6864 | 5116 |
| 14D | NA | 21195 | 11074 | 8692 | 6579 | 5321 |
| 16A | 16190 | 24972 | 12728 | 9705 | 7610 | 6079 |
| 16B | 17070 | 26500 | 13336 | 11395 | 8459 | 6788 |
| 16C | 20539 | 31872 | 17563 | 13280 | 9683 | 7881 |
| 16D | 14779 | 22609 | 11181 | 8734 | 6823 | 6122 |
| 24A | 25009 | 39719 | 21046 | 16509 | NA | 9736 |
| 24B | 21501 | 34636 | 17622 | 13602 | NA | 10070 |
| 24C | 29648 | 46341 | 22430 | 17959 | NA | 11802 |
| 24D | 24793 | 39983 | 19583 | 16136 | NA | 9986 |

Simulation Time per Vector

Appendix D

# Profiling Data

The following tables contain the raw profiling data for the six partition experiment. The first six tables contain the data for each separate partition, while the last table contains the aggregate sum. The data in each column are as follows:

1. The name of the subroutine.
2. The total time spent in each subroutine, measured in units of clock ticks (16.2mSec per tick).
3. The total number of calls to each subroutine.
4. The average time spent in each call. This is the quotient of the total time (expressed in mSec.) divided by the number of calls.
5. The percentage of the total time that was spent in each subroutine.
6. The percentage of the active simulation time spent in each subroutine. The active simulation time is the total time minus the idle time (time spent in `step`).

Subroutines with a "0" number of calls are library routines which were not recom-

piled with the profiling code.

| Subroutine | Time | No. Calls | mSec/Call | % Total | % Active |
|---|---|---|---|---|---|
| step | 172405 | 51 | 54763.941 | 50.04 | 0.00 |
| qldiv | 66391 | 6566045 | 0.164 | 19.27 | 38.56 |
| c_thev | 19665 | 659956 | 0.483 | 5.71 | 11.42 |
| cvtcond | 14475 | 4892172 | 0.048 | 4.20 | 8.41 |
| Handler | 10073 | 5011287 | 0.033 | 2.92 | 5.85 |
| lqmul | 9923 | 3144616 | 0.051 | 2.88 | 5.76 |
| muldiv | 7601 | 2925075 | 0.042 | 2.21 | 4.42 |
| sim_step | 6376 | 51 | 2025.318 | 1.85 | 3.70 |
| msg_poll | 6080 | 0 | 0.000 | 1.76 | 3.53 |
| new_val | 4918 | 199081 | 0.400 | 1.43 | 2.86 |
| main | 4781 | 0 | 0.000 | 1.39 | 2.78 |
| checkpt_nodes | 3991 | 3226 | 20.042 | 1.16 | 2.32 |
| lmul | 3823 | 0 | 0.000 | 1.11 | 2.22 |
| enque | 3143 | 335994 | 0.152 | 0.91 | 1.83 |
| make_clist | 2674 | 199081 | 0.218 | 0.78 | 1.55 |
| setin | 2225 | 16004 | 2.252 | 0.65 | 1.29 |
| check_inputs | 1330 | 376813 | 0.057 | 0.39 | 0.77 |
| uldiv | 1158 | 0 | 0.000 | 0.34 | 0.67 |
| checkpoint | 1042 | 3226 | 5.233 | 0.30 | 0.61 |
| cshare_make_clist | 630 | 25976 | 0.393 | 0.18 | 0.37 |
| rcmul | 591 | 219541 | 0.044 | 0.17 | 0.34 |
| charge_share | 503 | 25976 | 0.314 | 0.15 | 0.29 |
| cleanup_hist | 442 | 51 | 140.400 | 0.13 | 0.26 |
| lrem | 84 | 0 | 0.000 | 0.02 | 0.05 |
| check_overflow | 73 | 30776 | 0.038 | 0.02 | 0.04 |
| find | 60 | 16004 | 0.061 | 0.02 | 0.03 |
| msg_handler | 58 | 16208 | 0.058 | 0.02 | 0.03 |
| node_change | 9 | 1599 | 0.091 | 0.00 | 0.01 |
| sbrk | 9 | 0 | 0.000 | 0.00 | 0.01 |
| msg_free | 8 | 399 | 0.325 | 0.00 | 0.00 |
| msg_cons | 6 | 1650 | 0.059 | 0.00 | 0.00 |
| L0Send | 6 | 0 | 0.000 | 0.00 | 0.00 |
| msg_send | 4 | 1650 | 0.039 | 0.00 | 0.00 |
| malloc | 3 | 0 | 0.000 | 0.00 | 0.00 |
| msg_alloc | 2 | 1650 | 0.020 | 0.00 | 0.00 |

Profiling Data for Partition # 1

| Subroutine | Time | No. Calls | mSec/Call | % Total | % Active |
|---|---|---|---|---|---|
| step | 174648 | 51 | 55476.424 | 50.27 | 0.00 |
| qldiv | 63293 | 6276652 | 0.163 | 18.22 | 36.63 |
| c_thev | 18164 | 632654 | 0.465 | 5.23 | 10.51 |
| cvtcond | 13401 | 4669912 | 0.046 | 3.86 | 7.76 |
| Handler | 10086 | 5135188 | 0.032 | 2.90 | 5.84 |
| lqmul | 9543 | 3004391 | 0.051 | 2.75 | 5.52 |
| checkpt_nodes | 7787 | 6346 | 19.879 | 2.24 | 4.51 |
| muldiv | 6949 | 2785478 | 0.040 | 2.00 | 4.02 |
| sim_step | 6332 | 316 | 324.615 | 1.82 | 3.66 |
| msg_poll | 6124 | 0 | 0.000 | 1.76 | 3.54 |
| lmul | 5614 | 0 | 0.000 | 1.62 | 3.25 |
| new_val | 4819 | 221392 | 0.353 | 1.39 | 2.79 |
| main | 4626 | 0 | 0.000 | 1.33 | 2.68 |
| enque | 3201 | 351800 | 0.147 | 0.92 | 1.85 |
| make_clist | 2694 | 221392 | 0.197 | 0.78 | 1.56 |
| setin | 2363 | 17603 | 2.175 | 0.68 | 1.37 |
| checkpoint | 2019 | 6346 | 5.154 | 0.58 | 1.17 |
| check_inputs | 1395 | 398133 | 0.057 | 0.40 | 0.81 |
| uldiv | 1064 | 0 | 0.000 | 0.31 | 0.62 |
| cleanup_hist | 877 | 51 | 278.576 | 0.25 | 0.51 |
| cshare_make_clist | 599 | 2505 | 3.874 | 0.17 | 0.35 |
| rcmul | 531 | 218913 | 0.039 | 0.15 | 0.31 |
| charge_share | 479 | 25050 | 0.310 | 0.14 | 0.28 |
| restore_nodes | 392 | 265 | 23.964 | 0.11 | 0.23 |
| roll_back | 100 | 265 | 6.113 | 0.03 | 0.06 |
| lrem | 98 | 0 | 0.000 | 0.03 | 0.06 |
| check_overflow | 75 | 34899 | 0.035 | 0.02 | 0.04 |
| find | 59 | 17603 | 0.054 | 0.02 | 0.03 |
| msg_handler | 49 | 17794 | 0.045 | 0.01 | 0.03 |
| sbrk | 36 | 0 | 0.000 | 0.01 | 0.02 |
| msg_alloc | 6 | 902 | 0.108 | 0.00 | 0.00 |
| msg_free | 6 | 198 | 0.491 | 0.00 | 0.00 |
| malloc | 5 | 0 | 0.000 | 0.00 | 0.00 |
| node_change | 3 | 576 | 0.084 | 0.00 | 0.00 |
| msg_cons | 2 | 902 | 0.036 | 0.00 | 0.00 |
| L0Send | 2 | 0 | 0.000 | 0.00 | 0.00 |
| msg_send | 1 | 902 | 0.018 | 0.00 | 0.00 |
| settled | 1 | 61 | 0.266 | 0.00 | 0.00 |

Profiling Data for Partition # 2

| Subroutine | Time | No. Calls | mSec/Call | % Total | % Active |
|---|---|---|---|---|---|
| qldiv | 96095 | 9587394 | 0.162 | 29.18 | 41.10 |
| step | 95473 | 51 | 30326.718 | 28.99 | 0.00 |
| c_thev | 27643 | 959099 | 0.467 | 8.39 | 11.82 |
| cvtcond | 20807 | 7148518 | 0.047 | 6.32 | 8.90 |
| lqmul | 14565 | 4595117 | 0.051 | 4.42 | 6.23 |
| muldiv | 10689 | 4273260 | 0.041 | 3.25 | 4.57 |
| Handler | 9720 | 4938794 | 0.032 | 2.95 | 4.16 |
| sim_step | 8030 | 51 | 2550.706 | 2.44 | 3.43 |
| new_val | 6748 | 296440 | 0.369 | 2.05 | 2.89 |
| msg_poll | 5871 | 0 | 0.000 | 1.78 | 2.51 |
| setin | 4763 | 22980 | 3.358 | 1.45 | 2.04 |
| checkpt_nodes | 4677 | 3406 | 22.245 | 1.42 | 2.00 |
| enque | 4564 | 498914 | 0.148 | 1.39 | 1.95 |
| main | 4360 | 0 | 0.000 | 1.32 | 1.86 |
| lmul | 4180 | 0 | 0.000 | 1.27 | 1.79 |
| make_clist | 3662 | 296440 | 0.200 | 1.11 | 1.57 |
| uldiv | 1613 | 0 | 0.000 | 0.49 | 0.69 |
| check_inputs | 1422 | 426232 | 0.054 | 0.43 | 0.61 |
| checkpoint | 1100 | 3406 | 5.232 | 0.33 | 0.47 |
| cshare_make_clist | 970 | 39123 | 0.402 | 0.29 | 0.41 |
| rcmul | 813 | 321857 | 0.041 | 0.25 | 0.35 |
| charge_share | 683 | 39123 | 0.283 | 0.21 | 0.29 |
| cleanup_hist | 472 | 51 | 149.929 | 0.14 | 0.20 |
| lrem | 104 | 0 | 0.000 | 0.03 | 0.04 |
| check_overflow | 100 | 35256 | 0.046 | 0.03 | 0.04 |
| find | 86 | 22980 | 0.061 | 0.03 | 0.04 |
| msg_handler | 61 | 23140 | 0.043 | 0.02 | 0.03 |
| sbrk | 13 | 0 | 0.000 | 0.00 | 0.01 |
| node_change | 6 | 641 | 0.152 | 0.00 | 0.00 |
| msg_free | 3 | 159 | 0.306 | 0.00 | 0.00 |
| malloc | 3 | 0 | 0.000 | 0.00 | 0.00 |
| msg_cons | 2 | 692 | 0.047 | 0.00 | 0.00 |
| msg_alloc | 2 | 692 | 0.047 | 0.00 | 0.00 |
| settled | 2 | 51 | 0.635 | 0.00 | 0.00 |
| msg_send | 0 | 692 | 0.000 | 0.00 | 0.00 |

Profiling Data for Partition # 3

| Subroutine | Time | No. Calls | mSec/Call | % Total | % Active |
|---|---|---|---|---|---|
| qldiv | 105704 | 10520809 | 0.163 | 32.59 | 41.61 |
| step | 70274 | 51 | 22322.329 | 21.67 | 0.00 |
| c_thev | 30744 | 1052866 | 0.473 | 9.48 | 12.10 |
| cvtcond | 22494 | 7842552 | 0.046 | 6.94 | 8.86 |
| lqmul | 16116 | 5041376 | 0.052 | 4.97 | 6.34 |
| muldiv | 11559 | 4693488 | 0.040 | 3.56 | 4.55 |
| Handler | 9736 | 4948014 | 0.032 | 3.00 | 3.83 |
| sim_step | 8473 | 103 | 1332.647 | 2.61 | 3.34 |
| new_val | 7236 | 310824 | 0.377 | 2.23 | 2.85 |
| msg_poll | 5804 | 0 | 0.000 | 1.79 | 2.28 |
| checkpt_nodes | 5354 | 3887 | 22.314 | 1.65 | 2.11 |
| enque | 5021 | 532620 | 0.153 | 1.55 | 1.98 |
| setin | 4585 | 23045 | 3.223 | 1.41 | 1.80 |
| lmul | 4560 | 0 | 0.000 | 1.41 | 1.80 |
| main | 4493 | 0 | 0.000 | 1.39 | 1.77 |
| make_clist | 3866 | 310824 | 0.201 | 1.19 | 1.52 |
| uldiv | 1761 | 0 | 0.000 | 0.54 | 0.69 |
| check_inputs | 1501 | 444225 | 0.055 | 0.46 | 0.59 |
| checkpoint | 1341 | 3887 | 5.589 | 0.41 | 0.53 |
| cshare_make_clist | 1035 | 42618 | 0.393 | 0.32 | 0.41 |
| rcmul | 850 | 347888 | 0.040 | 0.26 | 0.33 |
| charge_share | 712 | 42618 | 0.271 | 0.22 | 0.28 |
| cleanup_hist | 536 | 51 | 170.259 | 0.17 | 0.21 |
| lrem | 106 | 0 | 0.000 | 0.03 | 0.04 |
| restore_nodes | 93 | 52 | 28.973 | 0.03 | 0.04 |
| check_overflow | 92 | 36748 | 0.041 | 0.03 | 0.04 |
| msg_handler | 84 | 23210 | 0.059 | 0.03 | 0.03 |
| find | 77 | 23045 | 0.054 | 0.02 | 0.03 |
| sbrk | 25 | 0 | 0.000 | 0.01 | 0.01 |
| roll_back | 19 | 52 | 5.919 | 0.01 | 0.01 |
| node_change | 12 | 1341 | 0.145 | 0.00 | 0.00 |
| msg_alloc | 8 | 1444 | 0.090 | 0.00 | 0.00 |
| msg_free | 8 | 335 | 0.387 | 0.00 | 0.00 |
| msg_send | 7 | 1444 | 0.079 | 0.00 | 0.00 |
| L0Send | 6 | 0 | 0.000 | 0.00 | 0.00 |
| msg_cons | 2 | 1444 | 0.022 | 0.00 | 0.00 |
| al_bytes | 2 | 167 | 0.194 | 0.00 | 0.00 |
| rollback_notify | 1 | 52 | 0.312 | 0.00 | 0.00 |
| malloc | 1 | 51 | 0.318 | 0.00 | 0.00 |
| settled | 0 | 51 | 0.000 | 0.00 | 0.00 |

Profiling Data for Partition # 4

| Subroutine | Time | No. Calls | mSec/Call | % Total | % Active |
|---|---|---|---|---|---|
| qldiv | 112574 | 11195769 | 0.163 | 34.85 | 41.55 |
| step | 52083 | 51 | 16544.012 | 16.13 | 0.00 |
| c_thev | 32214 | 1114184 | 0.468 | 9.97 | 11.89 |
| cvtcond | 24011 | 8383728 | 0.046 | 7.43 | 8.86 |
| lqmul | 16919 | 5354395 | 0.051 | 5.24 | 6.25 |
| muldiv | 12327 | 4989152 | 0.040 | 3.82 | 4.55 |
| Handler | 9915 | 4954787 | 0.032 | 3.07 | 3.66 |
| sim_step | 8676 | 113 | 1243.816 | 2.69 | 3.20 |
| new_val | 7603 | 323078 | 0.381 | 2.35 | 2.81 |
| msg_poll | 5914 | 0 | 0.000 | 1.83 | 2.18 |
| checkpt_nodes | 5521 | 4029 | 22.199 | 1.71 | 2.04 |
| enque | 5213 | 562673 | 0.150 | 1.61 | 1.92 |
| lmul | 4811 | 0 | 0.000 | 1.49 | 1.78 |
| setin | 4772 | 23745 | 3.256 | 1.48 | 1.76 |
| main | 4361 | 0 | 0.000 | 1.35 | 1.61 |
| make_clist | 4195 | 323078 | 0.210 | 1.30 | 1.55 |
| uldiv | 1820 | 0 | 0.000 | 0.56 | 0.67 |
| check_inputs | 4557 | 455497 | 0.162 | 1.41 | 1.68 |
| checkpoint | 1365 | 4029 | 5.488 | 0.42 | 0.50 |
| cshare_make_clist | 1171 | 44897 | 0.423 | 0.36 | 0.43 |
| rcmul | 967 | 365243 | 0.043 | 0.30 | 0.36 |
| charge_share | 848 | 44897 | 0.306 | 0.26 | 0.31 |
| cleanup_hist | 545 | 51 | 173.118 | 0.17 | 0.20 |
| find | 120 | 23745 | 0.082 | 0.04 | 0.04 |
| restore_nodes | 108 | 62 | 28.219 | 0.03 | 0.04 |
| msg_handler | 101 | 23908 | 0.068 | 0.03 | 0.04 |
| lrem | 94 | 0 | 0.000 | 0.03 | 0.03 |
| check_overflow | 79 | 38553 | 0.033 | 0.02 | 0.03 |
| roll_back | 31 | 62 | 8.100 | 0.01 | 0.01 |
| sbrk | 28 | 0 | 0.000 | 0.01 | 0.01 |
| node_change | 14 | 1281 | 0.177 | 0.00 | 0.01 |
| msg_send | 7 | 1394 | 0.081 | 0.00 | 0.00 |
| L0Send | 7 | 0 | 0.000 | 0.00 | 0.00 |
| msg_alloc | 6 | 1394 | 0.070 | 0.00 | 0.00 |
| msg_free | 5 | 341 | 0.238 | 0.00 | 0.00 |
| msg_cons | 3 | 1394 | 0.035 | 0.00 | 0.00 |
| malloc | 2 | 0 | 0.000 | 0.00 | 0.00 |
| rollback_notify | 1 | 62 | 0.261 | 0.00 | 0.00 |
| al_bytes | 0 | 175 | 0.000 | 0.00 | 0.00 |
| settled | 0 | 51 | 0.000 | 0.00 | 0.00 |

Profiling Data for Partition # 5

| Subroutine | Time | No. Calls | mSec/Call | % Total | % Active |
|---|---|---|---|---|---|
| qldiv | 125032 | 12343811 | 0.164 | 40.25 | 41.84 |
| c_thev | 37045 | 1224108 | 0.490 | 11.92 | 12.40 |
| cvtcond | 27094 | 9248812 | 0.047 | 8.72 | 9.07 |
| lqmul | 19018 | 5918752 | 0.052 | 6.12 | 6.36 |
| muldiv | 14468 | 5515133 | 0.042 | 4.66 | 4.84 |
| step | 11799 | 51 | 3747.918 | 3.80 | 0.00 |
| Handler | 10000 | 4811223 | 0.034 | 3.22 | 3.35 |
| sim_step | 9626 | 151 | 1032.723 | 3.10 | 3.22 |
| new_val | 8625 | 348289 | 0.401 | 2.78 | 2.89 |
| checkpt_nodes | 6478 | 4680 | 22.424 | 2.09 | 2.17 |
| msg_poll | 5816 | 0 | 0.000 | 1.87 | 1.95 |
| enque | 5771 | 610069 | 0.153 | 1.86 | 1.93 |
| lmul | 5560 | 0 | 0.000 | 1.79 | 1.86 |
| setin | 4942 | 23685 | 3.380 | 1.59 | 1.65 |
| make_clist | 4741 | 348289 | 0.221 | 1.53 | 1.59 |
| main | 4628 | 0 | 0.000 | 1.49 | 1.55 |
| uldiv | 2089 | 0 | 0.000 | 0.67 | 0.70 |
| check_inputs | 1721 | 476487 | 0.059 | 0.55 | 0.58 |
| checkpoint | 1570 | 4680 | 5.435 | 0.51 | 0.53 |
| cshare_make_clist | 1346 | 49596 | 0.440 | 0.43 | 0.45 |
| rcmul | 1092 | 403619 | 0.044 | 0.35 | 0.37 |
| charge_share | 893 | 49596 | 0.292 | 0.29 | 0.30 |
| cleanup_hist | 662 | 51 | 210.282 | 0.21 | 0.22 |
| restore_nodes | 188 | 100 | 30.456 | 0.06 | 0.06 |
| lrem | 112 | 0 | 0.000 | 0.04 | 0.04 |
| msg_handler | 103 | 23851 | 0.070 | 0.03 | 0.03 |
| find | 92 | 23685 | 0.063 | 0.03 | 0.03 |
| check_overflow | 75 | 40820 | 0.030 | 0.02 | 0.03 |
| roll_back | 34 | 100 | 5.508 | 0.01 | 0.01 |
| sbrk | 30 | 0 | 0.000 | 0.01 | 0.01 |
| malloc | 2 | 0 | 0.000 | 0.00 | 0.00 |
| al_bytes | 1 | 197 | 0.082 | 0.00 | 0.00 |
| msg_cons | 1 | 151 | 0.107 | 0.00 | 0.00 |
| msg_alloc | 1 | 151 | 0.107 | 0.00 | 0.00 |
| msg_send | 0 | 151 | 0.000 | 0.00 | 0.00 |
| rollback_notify | 0 | 100 | 0.000 | 0.00 | 0.00 |
| settled | 0 | 51 | 0.000 | 0.00 | 0.00 |
| msg_free | 0 | 30 | 0.000 | 0.00 | 0.00 |

Profiling Data for Partition # 6

| Subroutine | Time | No. Calls | mSec/Call | % Total | % Active |
|---|---|---|---|---|---|
| step | 576682 | 306 | 30530.224 | 29.14 | 0.00 |
| qldiv | 569089 | 56490480 | 0.163 | 28.75 | 40.57 |
| c_thev | 165475 | 5642867 | 0.475 | 8.36 | 11.80 |
| cvtcond | 122282 | 42185694 | 0.047 | 6.18 | 8.72 |
| lqmul | 86084 | 27058647 | 0.052 | 4.35 | 6.14 |
| muldiv | 63593 | 25181586 | 0.041 | 3.21 | 4.53 |
| Handler | 59530 | 29799293 | 0.032 | 3.01 | 4.24 |
| sim_step | 47513 | 785 | 980.523 | 2.40 | 3.39 |
| new_val | 39949 | 1699104 | 0.381 | 2.02 | 2.85 |
| msg_poll | 35609 | 0 | 0.000 | 1.80 | 2.54 |
| checkpt_nodes | 33808 | 25574 | 21.416 | 1.71 | 2.41 |
| lmul | 28548 | 0 | 0.000 | 1.44 | 2.04 |
| main | 27249 | 0 | 0.000 | 1.38 | 1.94 |
| enque | 26913 | 2892070 | 0.151 | 1.36 | 1.92 |
| setin | 23650 | 127062 | 3.015 | 1.19 | 1.69 |
| make_clist | 21832 | 1699104 | 0.208 | 1.10 | 1.56 |
| check_inputs | 11926 | 2577387 | 0.075 | 0.60 | 0.85 |
| uldiv | 9505 | 0 | 0.000 | 0.48 | 0.68 |
| checkpoint | 8437 | 25574 | 5.344 | 0.43 | 0.60 |
| cshare_make_clist | 5751 | 204715 | 0.455 | 0.29 | 0.41 |
| rcmul | 4844 | 1877061 | 0.042 | 0.24 | 0.35 |
| charge_share | 4118 | 227260 | 0.294 | 0.21 | 0.29 |
| cleanup_hist | 3534 | 306 | 187.094 | 0.18 | 0.25 |
| restore_nodes | 781 | 479 | 26.414 | 0.04 | 0.06 |
| lrem | 598 | 0 | 0.000 | 0.03 | 0.04 |
| check_overflow | 494 | 217052 | 0.037 | 0.02 | 0.04 |
| find | 494 | 127062 | 0.063 | 0.02 | 0.04 |
| msg_handler | 456 | 128111 | 0.058 | 0.02 | 0.03 |
| roll_back | 184 | 479 | 6.223 | 0.01 | 0.01 |
| sbrk | 141 | 0 | 0.000 | 0.01 | 0.01 |
| node_change | 44 | 5438 | 0.131 | 0.00 | 0.00 |
| msg_free | 30 | 1462 | 0.332 | 0.00 | 0.00 |
| msg_alloc | 25 | 6233 | 0.065 | 0.00 | 0.00 |
| L0Send | 21 | 0 | 0.000 | 0.00 | 0.00 |
| msg_send | 19 | 6233 | 0.049 | 0.00 | 0.00 |
| msg_cons | 16 | 6233 | 0.042 | 0.00 | 0.00 |
| malloc | 16 | 51 | 5.082 | 0.00 | 0.00 |
| al_bytes | 3 | 539 | 0.090 | 0.00 | 0.00 |
| rollback_notify | 2 | 214 | 0.151 | 0.00 | 0.00 |

Aggregate Profiling Data

# References


[1] M. Abramovici, Y. H. Levenel, and P. R. Menon, "A Logic Simulation Machine," *Proceedings of the 9th Annual Symposium on Computer Architecture,* 1982.

[2] T. L. Anderson, *The Design of a Multiprocessor Development System,* MIT Laboratory for Computer Science TR-279, 1982.

[3] R. E. Bryant, *A Switch-level Simulation Model for Integrated Logic Circuits,* MIT Laboratory for Computer Science TR-259, 1981.

[4] S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "Logic Circuit Simulator," *The Bell System Technical Journal,* **53**:8, pp 1451–1476, 1974.

[5] B. R. Chawla, H. K. Gummel, and P. Koziak, "MOTIS – An MOS Timing Simulator," *IEEE Transactions on Circuits and Systems,* **CAS-22**, December, 1975.

[6] J. T. Deutsch, and A. R. Newton, "A Multiprocessor Implementation of Relaxation Based Electrical Circuit Simulation," *Proceedings of the 21st Design Automation Conference,* 1984.

[7] R. H. Halstead, *et al.*, "Concert: The Design of a Multiprocessor Development System," in preparation.

[8] D. Jefferson, "Virtual Time," *Proceedings of the Parallel Processing Conference,* 1983.

[9] N. Jouppi, "TV: An nMOS Timing Analyzer," *Proceedings of the 3rd Caltech VLSI Conference,* 1983.

[10] B. W. Kernighan, and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal,* **49**:2, pp 291–307, 1970.

[11] U. Lauther, "A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation," *Proceedings of the 16th Design Automation Conference,* 1979.

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

[13] A. R. Newton, and A. L. Sangiovanni-Vincentelli, "Relaxation-Based Electrical Simulation," *IEEE Transactions on Electronic Design,* **ED-30**:9, September, 1983.

[14] J. Ousterhout, "Crystal: An Timing Analyzer for nMOS VLSI Circuits," *Proceedings of the 3rd Caltech VLSI Conference,* 1983.

[15] G. F. Pfister, "The Yorktown Simulation Engine," *Proceedings of the 19th Design Automation Conference,* 1982.

[16] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering,* **SE-1**:2, June, 1975.

[17] D. L. Russell, "State Restoration in Systems of Communicating Processes," *IEEE Transactions on Software Engineering,* **SE-6**:2, March, 1980.

[18] R. Saleh, and A. R. Newton, "Iterated Timing Analysis and SPLICE1," *Proceedings of the IEEE ICCAD Conference,* 1983.

[19] C. J. Terman, *Simulation Tools for Digital LSI Design,* MIT Laboratory for Computer Science TR-304, 1983.

[20] C. J. Terman, *User's Guide to NET, PRESIM, and RNL,* MIT Laboratory for Computer Science, September, 1982.

**CS-TR Scanning Project**
**Document Control Form**

Date : 4/27/95

**Report #** LCS-TR-333

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

## Document Information

Number of pages: 98(108-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page (2)
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): 2,4,6,12,74,52,68,84,96

Photographs/Tonal Material (by page number): ~~3~~

Other (note description/page number):

| Description : | Page Number: |
|---|---|
| IMAGE MAP (1) UNNUMBERED TITLE PAGE | |
| (2-98) PAGES #'ED 2-98 | |
| (99-105) SCANCONTROL, COVERS(2), SPINE, PRINTER'S NOTES, DOD (2) | |
| (106-108) TRGTS (3) | |

Scanning Agent Signoff:
Date Received: 4/27/95 Date Scanned: 5/5/95 Date Returned: 5/11/95

Scanning Agent Signature: Michael W. Cook

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>MIT/LCS/TR-333 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>Parallel Simulation of Digital LSI Circuits | | 5. TYPE OF REPORT & PERIOD COVERED<br>Master's Thesis<br>February 1985<br>6. PERFORMING ORG. REPORT NUMBER<br>MIT/LCS/TR-333 |
| 7. AUTHOR(s)<br>Jeffrey M. Arnold | | 8. CONTRACT OR GRANT NUMBER(s)<br>DARPA/DOD<br>N00014-83-K-0125 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>MIT Laboratory for Computer Science<br>545 Technology Square<br>Cambridge, MA 02139 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>DARPA/DOD<br>1400 Wilson Blvd.<br>Arlington, VA 22209 | | 12. REPORT DATE<br>February 1985<br>13. NUMBER OF PAGES<br>98 |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*<br>ONR/Department of the Navy<br>Information Systems Program<br>Arlington, VA 22217 | | 15. SECURITY CLASS. *(of this report)*<br>Unclassified<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*
Approved for public release, distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*
Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
Circuit simulation, parallel processing.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

**Integrated circuit technology has been advancing at a phenomenal rate over the** last several years, and promises to continue to do so. If circuit design is to keep pace with fabrication technology, radically new approaches to computer-aided design will be necessary. One appealing approach is general purpose parallel processing. This thesis explores the issues involved in developing a framework for circuit simulation **which exploits the locality exhibited by circuit operation to achieve a high degree of**

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

parallelism. This framework maps the topology of the circuit onto the multiprocessor, assigning the simulation of individual partitions to separate processors. A new form of synchronization is developed, based upon a history maintenance and roll back strategy. The circuit simulator PRSIM was designed and implemented to determine the efficacy of this approach. The results of several preliminary experiments are reported, along with an analysis of the behavior of PRSIM.

# Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

**Scanned**

**Date:** 5/5/1995

**M.I.T. Libraries**
**Document Services**