Copyright © 1993, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

LOGIC SYNTHESIS AND MASSIVELY PARALLEL COMPUTERS: TOOLS FOR SPEEDING-UP LOGIC SIMULATION

by

Gary A. Jones

Memorandum No. UCB/ERL M93/16

16 February 1993

LOGIC SYNTHESIS AND MASSIVELY PARALLEL COMPUTERS: TOOLS FOR SPEEDING-UP LOGIC SIMULATION

by

Gary A. Jones

Memorandum No. UCB/ERL M93/16

16 February 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Logic Synthesis and Massively Parallel Computers: Tools for Speeding-Up Logic Simulation

Gary A. Jones

University of California
Berkeley, CA 94720

Department of Electrical Engineering
and Computer Sciences

Abstract

Simulation continues to be a major tool in the design of digital circuits. With increases in design sizes and the relative simulation times, the need for better simulation performance grows. Many studies have been performed on methods to improve simulation performance, covering both software techniques and hardware acceleration methods. This report combines both ideas. On the software side, the concept is presented for using logic synthesis techniques to produce better implementations of a circuit for functional simulation. From a hardware perspective, this concept is investigated using a simulator running on a massively parallel SIMD computer. Synthesis tools are used to modify the functional description of a circuit to increase the parallelism and shorten the expected simulation time while mapping the description for execution on the parallel architecture.

Professor A. Richard Newton
Research Advisor

Acknowledgements

I would like to thank my research advisor, Prof. A. Richard Newton, for his guidance and support during my years at Berkeley. This work would not have been possible without his inspiration and encouragement. I also thank Prof. Alberto Sangiovanni-Vincentelli for his constructive criticism and timely reading of this work. My research was sponsored in part by the Semiconductor Research Corporation under contract numbers 91-DC-008 and 92-DC-008, by the Microelectronics Innovation and Computer Research Opportunities (MICRO) program through the state of California under grant number 91101 and by the National Science Foundation under grant number EMC-8419744. I am grateful for their financial support. It has been a pleasure working with the excellent computer resources provided by the UC Berkeley EECS CAD Group and made possible by grants from Digital Equipment Corporation. I am also grateful to MasPar Computer Corporation for access to their MP-1 massively parallel computer and the excellent software tools available with it.

Special thanks to Chris Lennard, Bill Lin, and Mark Noworolski for their friendship both in and out of school, to Andie Mishoe for the long distance counseling sessions, and to Jenny Ho for some spice and some fun when I needed the sanity check. Kenneth and Lisa Lee Rust could always be counted on for a balanced perspective on life. I owe additional debts to my friends in Berkeley who made their own contributions, but may never read this report: Chris Lennard, Mark Noworolski, Bill Lin, Jenny Ho, Debra Weir, Henry Sheng, Henry Chang, Lisa Guerra, Kia Cooper, Christi Clark and Shane Ahn among many. They may not recognize their contributions, but they certainly made my life more enjoyable and complete.

I can never repay my family for their love and support over the years. Without their constant encouragement and support I would not have come to Berkeley, much less attained the degree. I can not conceive of anyone better to owe this debt. I thank my parents, Opal and Elizabeth Jones and my sisters Karen, Vicki and Patti, for emotional and financial support. I also thank Vicki for more direct help in reading, discussion and revision of this report. I thank each of them and the remainder of my family for everything I am today.

Contents

Table of Contents ii

List of Figures iv

List of Tables v

1 Introduction 1
- 1.1 Synthesis for Simulation 2
- 1.2 Report Organization 3

2 Digital Simulation Background 5
- 2.1 Levels of Digital Simulators 5
  - 2.1.1 Circuit-Level Simulation 6
  - 2.1.2 Switch-Level Simulation 6
  - 2.1.3 Gate-Level Simulation 7
  - 2.1.4 Functional Simulation 7
  - 2.1.5 Behavioral-Level Simulation 7
  - 2.1.6 Mixed-Level and Mixed-Mode Simulation 8
- 2.2 Discrete-Event Simulation Terminology and Classifications 8

3 Parallel Simulation 11
- 3.1 Circuit Parallelism Studies 11
- 3.2 Parallel SIMD Simulators 12

4 The Massively-Parallel SIMD Machine Model 14
- 4.1 The Hardware Model 14
- 4.2 The Instruction Set 14
  - 4.2.1 Functional Computation Instructions 15
  - 4.2.2 Data Access and Transfer Capability 15
  - 4.2.3 Specialized SIMD Operations 15

5 Parallel Simulation Algorithm / Implementation 16
- 5.1 Simulation Algorithm 17
- 5.2 Gate to Processor Mapping 17

5.3 Implementation 18

6 Synthesis Tools for Improving Simulation Performance **20**

6.1 Introduction 20

6.2 Synthesis Framework 21

6.3 Synthesis Tools 22

6.3.1 Conversion to Simulation Model: Mapping vs. Decomposition 22

6.3.2 Simplification Algorithms 25

6.3.3 Timing Algorithms 26

7 Experimental Results **29**

7.1 Benchmark Circuits 29

7.2 Performance Metrics 29

7.2.1 Dynamic Metrics 30

7.2.2 Static Metrics 31

7.3 Results 32

7.3.1 Simulation Performance 32

7.3.2 Synthesis Improvement to Simulation 33

8 Conclusions and Future Work **42**

Bibliography **43**

List of Figures

1.1 *Conventional Approach to Discrete-Event Simulation* 3
1.2 *Synthesis Techniques Used for Discrete-Event Simulation* 4

2.1 *A Simple Network* 9
2.2 *Basic Discrete Event Simulation Algorithm* 10
2.3 *Flow Chart of Basic Discrete Event Simulation* 10

5.1 *Simple Compiled-Mode Simulation Algorithm for Massively-Parallel SIMD Machine* 18
5.2 *Implementation for Massively-Parallel Machine* 19

6.1 *100,000-input Parity Network: Two Inputs per Gate* 21
6.2 *100,000-input Parity Network: Nine Inputs per Gate* 22

7.1 *Contention on a Single Input Fetch* 30
7.2 *Contention Time Comparison* 31
7.3 *TRISC Gates* vs. *Logic Level* 34
7.4 *TRISC Active Processors* vs. *IPC Step* 35
7.5 *Benchmark Processor Utilization* 36
7.6 *Synthesis Optimization for MP Simulation* 37

List of Tables

6.1 Comparison of Technology Mapping and Decomposition. 25

7.1 Performance Results for the MP Simulator. 33
7.2 Results for **simplify**. 38
7.3 Results for **full_simplify**. 39
7.4 Results for **collapse**. 39
7.5 Results for **speed_up**. 40
7.6 Results for **reduce_depth**. 40
7.7 Results for **reduce_depth** with simplification. 41

Chapter 1

Introduction

Computer-Aided Design (CAD) systems aid designers of integrated circuits through the verification of designs at all levels and the automatic synthesis of designs from a behavioral description to silicon. The primary goal of CAD systems is to produce near optimal implementations of a correct design as quickly as possible. Performance of a CAD system is crucial: the rapid growth of the integrated circuit market requires quick, correct designs in order to maintain competitiveness.

Integrated circuits are usually designed with the help of validation tools to guarantee correctness. After design, circuit descriptions are optimized using logic synthesis tools to minimize delay, minimize area, or some combination. The output of the synthesis is a netlist which is used to create the fabrication masks.

Simulation is the primary method for validation used in designing circuits. While other validation methods such as timing verification are useful tools in designing circuits, simulation is the most important method of assuring a design's correctness. Emerging techniques including formal verification and correct-by-construction techniques (silicon compilation) are reducing the need for extensive simulation. However, these techniques are only as valid as the input given. If the input to a formal verification tool or a correct-by-construction program has not been validated, the resulting design may not be what was intended. Formal specification can never fully replace simulation. Many times incorrect systems are specified and built because inadequate simulation was performed.

While simulation is necessary, it is also a hindrance. Simulation consumes much of the designer's effort and even more of the computational effort in designing a digital system. As systems become larger, the simulation costs can only grow. With larger systems, the number of interconnections between components greatly increases the amount of validation needed. Tomorrow's

simulators must be fast and must be able to handle the larger circuits being designed.

In general, circuit designs may be described at a variety of levels - functional, logic, etc. The description of a circuit's behavior is input to the simulator along with a collection of test cases. Most simulations systems consist of two general steps - the mapping of the description (input/output specification of a circuit) to the data structures used by the simulation engine and the simulation itself (that is, executing the simulation with the specified input) run on the simulation engine. During the mapping phase the description is also often optimized to reduce the run time of the simulation. As noted earlier, simulator performance is extremely important since simulation is considered the bottleneck in most integrated circuit designs.

In this report the simulation system attempts to achieve faster simulation through the use of synthesis tools to do the mapping of functional descriptions to the simulation format and a massively-parallel machine as the simulation engine.

1.1 Synthesis for Simulation

Conventional discrete-event simulation techniques include gate-level, RTL, behavior, and system-level. Current simulation tools are becoming obsolete as electronic systems rapidly become more complex. Even so-called behavioral simulation, if it is to maintain the precision required by many designs, does not provide sufficient speed-up over logic simulation to do the job.

In these conventional systems all design descriptions are ultimately evaluated at the gate-level, so the potential for speed-up is restricted. Since conventional mainframes and workstations are Von Neumann machines and implement only binary data types directly in the hardware, the final binary file produced by the compiler is really a logic-level description. The datapath of the simulation engine (workstation, mainframe) performs simple operations (and, or, etc.) and some special support is provided for the number datatype via the floating point hardware. In reality, the C-compiler (FORTRAN or ADA, in some cases) of the workstation is "synthesizing" a logic-level description of the system behavior which is intended to have the same functional characteristics as a real implementation. This description is intended to simulate quickly on a Von Neumann computer.

This gate-level evaluation is illustrated in Figure 1.1, where the levels of design abstraction are shown on the left. The Von Neumann computer is shown as a gate-level implementation since the dominant abstract data type it directly implements is the binary "bit" on wires in the main processor.

The key idea of using synthesis for simulation is to replace the general-purpose symbolic

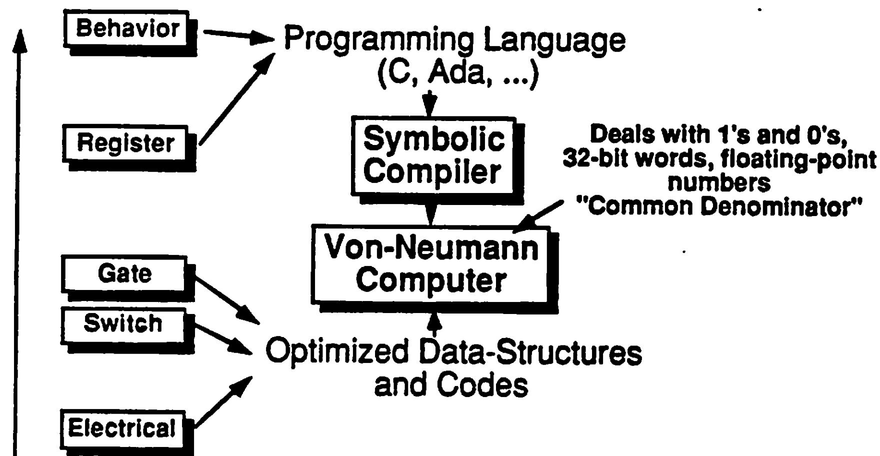

Figure 1.1: *Conventional Approach to Discrete-Event Simulation*

C-compiler (which maps a behavioral description to the format of the simulation engine) with effective synthesis tools which understand the special properties of the behavioral descriptions of digital electronic systems. These tools can manipulate the description from a mathematical and global point of view, while maintaining its external behavior, so that it simulates quickly. In addition to using synthesis to minimize the final area or delay time of the hardware implementation of the design, synthesis tools can be used early in the design cycle to improve the simulation speed (Figure 1.2).

Once synthesis tools are used to compile for simulation, they can also be used to retarget descriptions to other hardware platforms for improved simulation, such as massively-parallel single-instruction multiple-data (SIMD) machines, or even FPGA-based emulation engines, giving designers a wide range of price/performance options. This project targets the massively-parallel SIMD computer as the simulation engine. The two primary goals of the project were to evaluate the usefulness of synthesis tools to map behavioral descriptions to a simulation engine format and to evaluate the performance improvements of the simulation engine using SIMD architectures.

1.2 Report Organization

An introduction to digital simulation, including terminology and general types of simulators is presented in Chapter 2. Discrete event simulation algorithms and implementation techniques

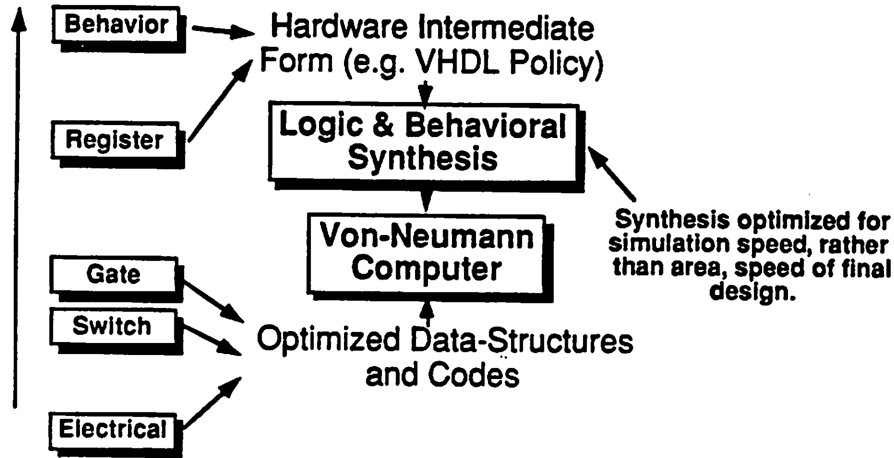

Figure 1.2: *Synthesis Techniques Used for Discrete-Event Simulation*

are described and some example simulators are reviewed. In Chapter 3 the potential parallelism in circuits and some previous work in parallel digital simulation is summarized. Previous work in the application of simulation techniques to various target hardware platforms is presented. A machine model for the massively-parallel SIMD computer in Chapter 4, which is the basis for the study of techniques in the remainder of this report.

A general simulation algorithm and implementation is presented in Chapter 5, which serves as the basis for investigating the use of synthesis tools and techniques as a preprocessor for parallel simulation. The synthesis tools and framework, experimental results, and conclusions are presented in Chapter 6, 7 and 8, respectively.

Chapter 2

Digital Simulation Background

Before presenting the simulation approach chosen for implementation, an understanding of the basic model of discrete-event simulation and its terminology is necessary. An overview of digital simulation and some necessary background is presented in this chapter. In the first section, the different levels of circuit simulators are defined. The discrete-event model and simulation classifications are defined in the final section.

2.1 Levels of Digital Simulators

There are many levels for modeling the behavior of a digital circuit. At the physical level, the silicon, metal, etc. may be treated as simple devices such as transistors with timing behavior derived from the physical components. Transistors may be modeled as simple on-off (digital) switches with special timing parameters to approximate the behavior of the physical elements of the transistors. Above this switch model, the transistors may be grouped into functional blocks implementing Boolean logic functions. These logic blocks, or gates, retain the functionality but are a less accurate model of the underlying physical elements. In the same way, gates can be grouped into behavioral or functional blocks. These blocks model the functions, but not the individual behavior of the underlying transistors. In each case the physical design is modeled at a more abstract and less physically accurate behavioral level. These models are often used to specify a design at a high level of abstraction. The design is then refined to successively lower abstractions, eventually leading to a physical design specification. Simulation tools are available to verify designs at every level of abstraction. These design levels translate to five types of digital simulation: circuit-level, switch-level, gate-level, functional-level and behavioral-level simulation. Techniques for simulation and

example simulators of each type are presented in the following sections.

2.1.1 Circuit-Level Simulation

Circuit-level simulation is the most accurate simulation of a circuit's behavior. The circuit is modeled as transistors, resistors and wires. The behavior of these elements is determined by their physical geometry and the technology in which the circuit is built. From this basis, a set of mathematical equations can be derived to represent this behavior. The state of any node in the circuit can be found by solving these equations. Detailed behavior provided by this level of simulation is essential to verify critical parts of a design. The computational expense of this detailed solution is too high for general simulation of large designs. Circuit-level simulators such as *SPICE* [Nag75] and *CAzM* [Erd89] are feasible for simulating up to 10,000 transistor networks. Today designs easily eclipse this size limit. To handle larger designs, simulation accuracy is traded against computational complexity. Investigation in reducing the computational complexity at the circuit level has concentrated on relaxation techniques [NSV83]. A simple model of the circuit elements may be chosen for simulation giving less accurate behavior modeling at a greatly reduced computational cost.

2.1.2 Switch-Level Simulation

In digital circuit design a transistor can be modeled as a simple switch with an acceptable loss of accuracy. For better analysis of critical portions of the circuit, circuit-level simulation may still be used for small pieces of the design. Switch-level simulators model the entire circuit as a collection of transistors and wires. The wires are generally modeled as idealized, zero-delay conductors. The transistors are modeled as switches with a simple delay model such as unit-delay in switching. More complex delay information may often be included if it is available. Some example switch-level simulators are *ESIM* [Ter83], *MOSSIM* [Bry84], and *COSMOS* [Bry87]. The circuit simulation involves solving simplified equations based on approximate circuit theory. Many switch-level simulators incorporate more accurate timing at the cost of additional computation time. Some also allow the user to specify increased levels of precision in modeling the element behavior. In spite of the loss of accuracy from circuit-level simulation, the timing information at the switch level may be sufficient to detect timing problems such as hazards, glitches and race conditions.

2.1.3 Gate-Level Simulation

Gate-level simulators model circuit elements at the gate rather than transistor level. The representation of a group of transistors as a simple Boolean logic gate greatly reduces the number of models to be evaluated and thus the total computation time for the simulation. Rather than equations for the voltage and current levels at nodes in the circuit, a set of Boolean logic equations represent the circuit behavior. The total simulation is more efficient due to the reduced number of equations and the simplicity of the basic logic computations. The circuit is represented as a collection of logic gates and the connecting wires. Gate-level simulators frequently support only a small set of functions: *and, nand, or, nor* and *dff.* A delay value is assigned to each gate, reducing the timing information as compared to switch-level simulation. Some simulators assign fixed or unit delays to each gate while others incorporate additional information about capacitance or fanout into the delay models. *HILO* [Gen85] and *THOR* [SB87] provide gate-level modeling for simulation.

2.1.4 Functional Simulation

Functional-level, or register-transfer-level, simulation is abstracted another step from the gate level. The logic gates are grouped into functional blocks of combinational and sequential components. The connections between these components are no longer restricted to wires. Related wires or bits may be grouped into ordered sets of words or buses. The simulation consists of a set of statements describing transfers of data between functional blocks and arithmetic operations on this data. The higher level of abstraction in functional-level simulation allows functional verification of a large design in a reasonable amount of computation time. This functional testing does not include the ability to catch subtle errors such as races, hazards and critical timing constraints. These timing errors can only be detected at the lower levels presented above.

2.1.5 Behavioral-Level Simulation

Behavioral-level simulation is very similar to functional-level simulation. Both represent the circuit design by a set of blocks with functions specified directly by the designer. The main difference is the blocks correspond directly to hardware blocks in functional-level simulation. While behavioral-level descriptions duplicate behavior, but not necessarily the structure of the implementation. The behavioral-level units are generally described using a hardware description language such as VHDL. Similar to software programming languages these descriptions can be compiled and executed to emulate the operation of the design specification. Like functional-level

simulation, behavioral-level simulation is very efficient for verifying the high-level operation of the design, but does not include the information to detect low-level timing or design errors.

2.1.6 Mixed-Level and Mixed-Mode Simulation

Mixed-level simulation combines two different levels of simulation, such as switch and functional, in a single simulator. These are combined to allow different portions of a design to be tested at different levels. Critical pieces of the design can be simulated for detailed timing behavior at the switch-level, while less critical pieces only have their high-level behavior verified. Computation and precision tradeoffs are made in a single simulation run. *LDVSIM* [Bri89] and *Lsim2* [CE75] are examples of mixed-level simulators.

Where mixed-level simulation blends two different levels of abstraction for a single simulation, mixed-mode simulation mixes computational techniques, such as direct and relaxation methods, in a single simulation. Mixed-mode simulators such as *SPLICE* [New79] and *SAMSON* [SD80] improve simulation performance by their dynamic choice of algorithm.

2.2 Discrete-Event Simulation Terminology and Classifications

Figure 2.1 represents a simple circuit comprised of three gates. A change in one of the inputs may cause a change in one of the internal nodes which, in turn, may cause a change at the output. Simulation's role is to determine the effects of changes at the inputs and involves not only determining the values of the internal nodes, but also the time at which changes occur at the nodes.

The basic model of discrete-event simulation has a *clock* representing the current time in the system and an *event queue* consisting of events that will happen in the future relative to the clock. When a node changes state because of a change in the output of a simulated gate, the node change cannot be directly applied to the node until the appropriate time. A simulation *event* consists of a new value for a node and the time at which the change will occur. For example, assume an event has been placed on the event queue for the circuit input in Figure 2.1. When the clock reaches the time specified in the event, the event is dequeued and the new gate output value is calculated using the new input value. The new output value with the appropriate time is now an event for each of the gate fanout nodes. Each of these gates must be evaluated, potentially generating more events. The simulation continues until the event queue is empty or some maximum time value is reached. This basic discrete-event simulation algorithm is given in Figure 2.2 and depicted in Figure 2.3.

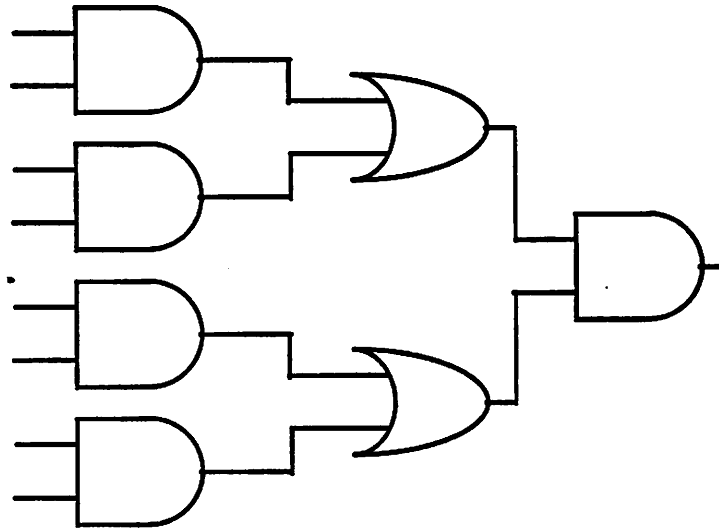

Figure 2.1: *A Simple Network*

The designers of HSS [B+87] defined several classifications for simulators which are of interest for discrete-event simulation. A simulation algorithm can be event-driven or oblivious. In oblivious simulation every gate is evaluated at every time step. In event-driven simulation only those gates whose input has changed are evaluated. Additionally, a simulation implementation can be compiled or interpretive. In interpretive simulation a data structure representing the circuit network is constructed. A central scheduler iterates over the simulation time, calling procedures to evaluate the network. In compiled simulation a customized program is produced which simulates the network. The simulation iterations of the interpretive scheduler are effectively unrolled to produce a straight line program with direct data addressing. This reduces the overhead of the simulation in traversing the network data structure. HSS4 [B+87] is an example of a compiled event-driven logic simulator and SSIM [WHPZ87] is an example of an interpretive oblivious simulation.

```
while ( events remain ) {
    take earliest event;
    update time;
    modify node;
    evaluate affected gates;
    enqueue any events determinde by evaluation;
}
```

Figure 2.2: *Basic Discrete Event Simulation Algorithm*

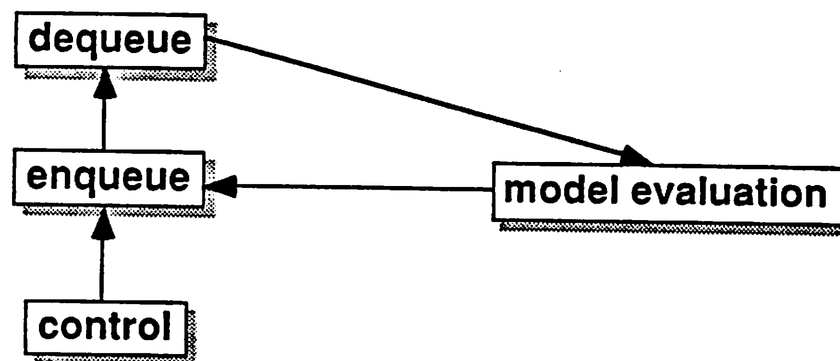

Figure 2.3: *Flow Chart of Basic Discrete Event Simulation*

Chapter 3

Parallel Simulation

Previous studies have tried to measure the potential of using parallel approaches for digital simulation. These studies, investigating circuit characteristics for parallel simulation, are reviewed in the next section. Previous work on simulators for massively parallel single-instruction multiple-data (SIMD) machines is reviewed in the final section.

3.1 Circuit Parallelism Studies

Many studies have investigated the feasibility of massively parallel simulation, which depends on the existence of large amounts of parallelism in the simulation model. In a study of potential parallelism in logic simulation [Fra86] gave very promising theoretical estimates. Similar studies ([BS88], [SB88] and [Won86]) have indicated small degrees of parallelism and forecast accordingly small speedups from parallel simulation.

Frank [Fra86] determined the theoretical speedup of simulating digital designs of up to 20,000 transistors. The author proposed a theoretical architecture consisting of unlimited numbers of processors connected with a network for instantaneous interprocessor communication. The instruction of the processors is optimized for switch-level simulation. The multiprocessor organization for switch-level simulation consists of a single simulation instruction per processor allowing unlimited operands per instruction, eg. unlimited inputs and outputs for nodes in the circuit. This proposed multiprocessor architecture was simulated on a uniprocessor machine. The resulting theoretical speedups were as high as 200 times over a simulation for a single processor. A more realistic simulation of the architecture using 64 processors obtained speedups of 28 times with no communication costs considered and 12 times with communication costs included. Frank concluded

that there is parallelism to exploit in switch-level simulation. He also concluded that the parallelism does not generally increase with the circuit size and average parallelism will be relatively limited even for large circuits.

A number of statistics on logic simulation were collected by Wong, *et al* [Won86] to identify the potential for increased simulation performance. The statistics collected include the number of events associated with each simulation component, the number of events in the event queue, times between events in the queue and queue activity. The main statistic of concern for predicting parallelism is the number of simultaneous events for queue activity. From their simulation of example circuits of less than 8,000 transistors the authors concluded that relatively few events occur in parallel. Further, they concluded that the amount of parallelism scales with the size of the circuit, thus offering opportunities for exploiting parallelism in larger circuits.

Soule and Blank collected statistics similar to Wong, *et al*'s in [SB88]. These statistics were collected on circuit designs specified at four description levels from gate to behavioral level. The circuits were simulated and analyzed for exploitable parallelism. The maximum speedup was measured using an "ideal" parallel environment model with no cost for processor memory contention and synchronization. Analyzing the simulation for up to 1,000 parallel processors they found a maximum speedup of 100 times for most circuits. For most circuits the maximum predicted speedup is only 5 times, with little correlation between speedup and design size.

A study by Bailey and Snyder [BS88] was expressly targeted to measure parallelism in CMOS circuit designs. They also used the event queue activity to measure the parallelism in the designs being simulated. Simulating a small number of circuits of up to 27,000 transistors, they obtained a maximum speedup of 25 times. The percentage of parallel activity in the circuits ranged from 0.04% to only 2.9% of the total design size. For their benchmark circuits, the amount of parallelism often decreased with increased circuit size.

3.2 Parallel SIMD Simulators

A parallel simulator implementation should give a more accurate performance measure for exploiting parallelism than a static analysis of serial simulation runs. Following the direction of this research report, a number of simulators implemented on massively parallel SIMD machines are presented below.

A relaxation-based circuit-level simulator was implemented by Webber, *et al*, [WSV87]. This simulator produced good results but used parallelization of circuit analysis techniques which

do not readily extend to parallel logic simulation.

A data parallel version of the switch-level simulator COSMOS was implemented by Bryant for a massively parallel SIMD machine [Bry88]. The entire switch-level model is replicated onto each processing element with a separate input vector evaluated on each. Using a 32,768 processor machine the simulator runs up to 33,000 times faster than a sequential simulator on a uniprocessor workstation.

The COSMOS switch-level simulator was extended for general parallel simulation on a SIMD machine by Kravitz and Bryant [KBR89]. The COSMOS algorithm decomposes a transistor circuit into a series of Boolean equations. These equations form a set of data independent modules which can be partitioned onto the massively parallel machine for maximum parallelism. From their investigations, they concluded that sufficient parallelism is available for speedup through massively parallel evaluation of the Boolean modules. The parallel implementation of COSMOS took twice as long to simulate one benchmark circuit and half as long to benchmark the second circuit, as the same simulation on the regular COSMOS simulator. The authors concluded that the one limiting factor for massively-parallel simulation was the interprocessor communication times on the SIMD machine.

The final example of parallel SIMD simulators is the VHDL simulator implemented by Vellandi [Vel90] for the Connection Machine. Vellandi developed a tool for restructuring the behavioral-level VHDL model functions into fine-grained computational units. Using this tool with a parallel SIMD simulator implementation she studied the dynamic behavior of the parallel simulation. From these studies she found an average component activity level on the SIMD machine of 25.2% with a maximum of 36.0% active and minimum of 6.7% active. The final speedups ranged from 1.4 to 10.4 with an average of 3.4 for their examples. These speedups were only achieved when the simulation support functions (i.e., event queue routines) were parallelized. Although the program restructuring techniques exposed a higher degree of parallelism than reported by previous studies, the results for parallelizing only the model evaluations was disappointing. Vellandi concluded that the potential exists for exploiting the parallelism of the circuits but a hardware platform other than the SIMD machine was necessary for real speedups.

Chapter 4

The Massively-Parallel SIMD Machine Model

4.1 The Hardware Model

The massively-parallel computer model considered contains thousands of processors, typically 1,024 to 65,536 processors. All the processors execute a single command in parallel on their own data in a single instruction, multiple data (SIMD) fashion. The processing elements which consist of the processor and its private local data space are conceptually arranged in a two dimensional array. There is no instruction space associated with an individual processor since instructions are provided by a control unit for the entire processor array. A connection to a sequential support machine is required for loading the initial circuit information, as well as providing inputs to and outputs from the simulation. Prime examples for this model are the Connection Machine [Hil86] and the MasPar machine [Bla90], [Nic90].

4.2 The Instruction Set

The types of operations which must be provided by the machine's instruction set to fully support the simulation are presented in the following sections.

4.2.1 Functional Computation Instructions

Arithmetic operations must be provided to evaluate the simulation models. Boolean operators are required for the evaluation of arbitrary logic functions. A small set of operators, such as AND, OR, INVERT, and perhaps XOR is sufficient. In addition, a number of basic arithmetic operations are required for the basic simulation implementation.

4.2.2 Data Access and Transfer Capability

Several levels of data access and transfer capabilities are required for the complete simulation implementation. For high-level programming, flexibility of the simulation memory and data addressing modes are a concern. Indirect memory addressing modes are necessary to support table lookup simulation methods, or interpretive simulation using extensive data structures to represent the circuit network. Indirect addressing modes allow different processing elements to access data in unique local data locations. This translates to the capability for pointer dereferencing in programming languages such as the C programming language. There are two levels of memory access instructions necessary for the particular arrangement of the SIMD architecture. These are required for the exchange of data between individual processing element's local memory, and the exchange of data between local processing element memory and the supporting 'front-end' machine. The access to the support machine is used to load the static circuit data, such as the network configuration, and to exchange information concerning circuit inputs and outputs during the simulation execution. The remaining instruction requirement is a communication instruction for moving data between processing elements. An instruction must be available to allow random processor-to-processor data transfer during the simulation. A single communication instruction using a global communication network is sufficient, although other instructions may be available for faster transfer of data for certain arrangements of data and processors.

4.2.3 Specialized SIMD Operations

The simulation program is to be executed on data across the entire processor array, even though an individual simulation step may only apply to data on a small portion of the total processors. A selection mechanism for the enabling/disabling of execution on selected processors is necessary for conditional execution of instructions. This instruction allows the use of conditional branches and loops in the higher level languages used for programming the simulation implementation.

Chapter 5

Parallel Simulation Algorithm / Implementation

The simulation approach presented in this report is an oblivious, interpretive approach. As they have been described in the previous chapters, this means that a data structure is generated which characterizes the static structure of the network to be simulated. This network is then traversed for each set of inputs to generate the simulation results, independently of the input data. While this would seem a very poor choice of algorithm / implementation for a sequential simulator, this is an appropriate choice for the parallel simulation study of this report. Both choices correspond to a trade-off of overhead and computational complexity for simplicity. In this case, the simple choice is more helpful in studying the usefulness of synthesis tools for improving the simulation performance.

The oblivious evaluation approach, seemingly resulting in evaluation of unchanging data, is appropriate for the massively parallel architecture. With hundreds or thousands of modules being evaluated in parallel, the likelihood of network activity is greatly increased for each evaluation step. The interpretive implementation choice is also driven by the special characteristics of the SIMD architecture. Since the simulated time traversal on a parallel execution model causes the evaluation of many logic models in parallel, the efficiency of the method as measured by overhead operations versus model evaluation operations is greatly increased. The use of a data structure to represent the circuit network, the interpretive approach for sequential architectures, can be viewed as the obvious extension of the compiled implementation for parallel architectures. The fact that the network does not change during simulation allows the use of an algorithm that efficiently simulates the network,

just as in compiled simulation for sequential architectures.

These choices are particularly useful in the examination of improvements in simulation performance between circuit instantiations. Using the simpler algorithm and implementations, clear relationships can be established between static circuit and network characteristics and the simulation performance. These network characteristics can then be used as the target parameters for optimization by the synthesis tools. With the simple simulation approach, the improvement to simulation should be easier to characterize and optimization tradeoffs more obvious. From previous studies, performance improvements should be possible through extensions of the basic simulator. The parallelizing of the simulation support routines, as in [Vel90], would be expected to improve the performance regardless of the execution approach chosen. Therefore, the absolute performance of the parallel simulator may be worse than possible with more complex simulation choices, the usefulness as a measure of synthesis for simulation is not affected.

5.1 Simulation Algorithm

The circuit is simulated using a simple evaluation loop (Figure 5.1). The simulation complexity depends on the static characteristics of the logic network. The gate functions and the communication patterns are defined by the synthesized logic network and are not dependent on the simulation input values. The gate output computation is performed on all processors in parallel, so the time required is negligible. The major cost of the logic simulation on the massively-parallel machine is the time to "fetch" the input values for the logic function. The communication time is, therefore, a function of the maximum number of inputs for any gate in the network.

5.2 Gate to Processor Mapping

The input to the simulator is a logic network describing the next-state logic and consisting of a set of logic gates and their interconnections. The logic network is level-ordered from the circuit inputs, and then mapped onto the processor array. The network is mapped starting with the primary inputs continuing by increasing logic level of the gates.

Since the mapping of gates to processors preserves the level order of the original circuit, the gate evaluation sequence can be performed for each successively increasing level of gates. Given N logic levels in the network, the entire circuit must be evaluated N times per simulation clock to propagate the input vector values through the network. The simulation results for that

```
foreach ( clock time in the simulation ) {
     foreach (level of logic) {
          forall ( gates at this level) {
             foreach ( input of the gate) {
                  fetch input value  from previous level;
             }
             compute gate output;
          }
     }
     set input state = output state;
     save/display selected state variables;
}
```

Figure 5.1: *Simple Compiled-Mode Simulation Algorithm for Massively-Parallel SIMD Machine*

simulation clock can then be saved from the output vector. This algorithm results in two parameters to optimize during synthesis for improved simulation performance. The number of logic levels in the circuit is the main parameter for minimization. Next, the number of inputs to gates in the network should be minimized to reduce the amount of data transfer between levels.

5.3 Implementation

The simulator has been implemented on the MasPar massively-parallel machine [Nic90]. The MasPar machine is a general purpose SIMD machine with up to 16,384 processing elements (PEs). Each PE consists of a 4-bit processor and 16K bytes of local data memory. As a general purpose machine, the MasPar processors support a full RISC instruction set. In addition to the Boolean operations and data transfer instructions required by our model massively-parallel simulation machine, the MasPar instruction set offers additional instructions which may be exploited for faster simulations. The MasPar instruction set includes arithmetic operations for the manipulation of numbers and other abstract data types. It also provides an 'xnet' command for transfer of data between physically close processing elements. Xnet uses a mesh interconnection network, instead of the global router network, to transfer data. This allows faster transfers, but is restricted to the local interconnections and will not work for transfers between random groups of processors.

The simulator consists of three main phases as shown in Figure 5.2. The simulation input

is a combinational logic description of the next-state logic network, as described in Section 3. In the first phase ($\phi 1$), the MIS logic synthesis system [BR+87] is used to produce a functionally equivalent network optimized for simulation on the MasPar machine. This functional equivalence holds for the external behavior, as seen at the input vector (primary inputs and state outputs) and the output vector (primary outputs and state inputs). Using the synthesis tools available in MIS, alternate implementations of the circuit are generated. These tools optimize for area and speed of the final circuit, but not directly for simulation. The circuit is also mapped to a library of logic gates for simulation on the massively-parallel machine. The MIS technology mapping algorithms, again oriented for goals other than simulation, are used for this mapping. As described previously, the "synthesized" logic network is level-ordered and mapped for simulation on the processor array. The mapped network description is converted to a data structure containing the information required for evaluation on the individual processors. The data is then written to a file for loading to the processor array. These steps are performed on the local workstation, in this case a DECstation 5000. The second phase ($\phi 2$) is performed on the MasPar machine. The network representation is loaded onto the parallel processor array by the simulation support routines. The simulation code is then executed for the simulation input patterns on the MasPar SIMD machine in phase three ($\phi 3$).

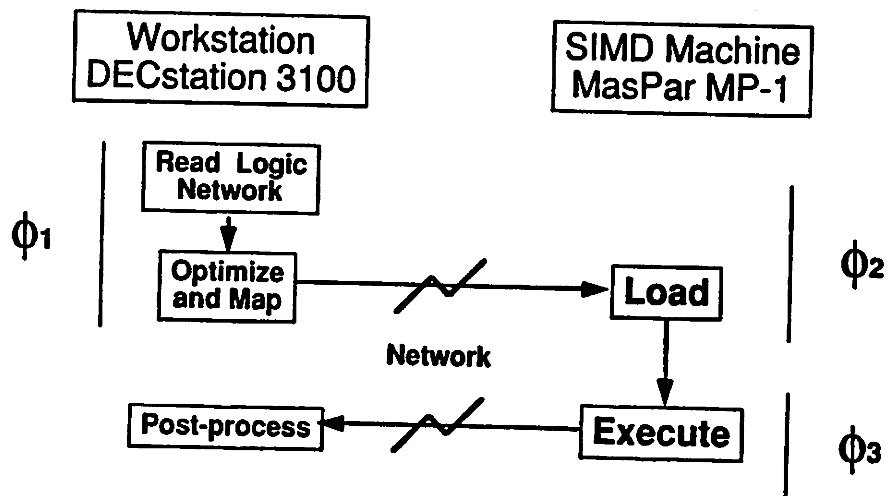

Figure 5.2: *Implementation for Massively-Parallel Machine*

Chapter 6

Synthesis Tools for Improving Simulation Performance

6.1 Introduction

Regardless of the initial level of design description (behavioral, RTL, logic) on the computer that must execute the simulation (in this case, a massively-parallel machine), the description consists of a collection of stored "bits" and a collection of combinational next-state and output functions. Again, at the level of the target simulation engine, these operations are expressed in terms of the fundamental operations available in the machine instruction set. For most descriptions, these instructions are dominated by the logic operators but may include more abstract operations on bytes or full words (e.g. arithmetic operations). So the target low-level model for the synthesis-for-simulation compiler is a state vector containing the present-state and primary input values of the description (R1), a combinational next-state network containing simple logic gates and a state-vector to receive the next-state and primary output values (R2). This model can be used to represent any collection of interacting synchronous or fundamental-mode asynchronous descriptions, derived from any level of abstraction.

As an example, consider the parity network shown in Figure 6.1. The parity network was chosen because of its regularity and its worst-case "don't care" property. In real examples, different forms of the same logic function are radically different due to the ability to exploit local don't cares. In this case, the network has 100,000 primary inputs, but only a single output. In Figure 6.1,the network has been implemented as a tree of 2-input XOR gates, with the final gate collecting any

outputs still to be combined. The number of stages of logic, as well as the number of inputs per gate, can be varied and still produce the same output. In Figure 6.2, a 9-input per gate implementation of the same function is shown. The range of possible implementations is particularly important for evaluation on a massively-parallel machine. The major cost of parallel logic simulation is the communication time for transferring data between processors, which overshadows the logic function evaluation time. Therefore, the simulation time will vary according to the structure of the circuit implementation chosen.

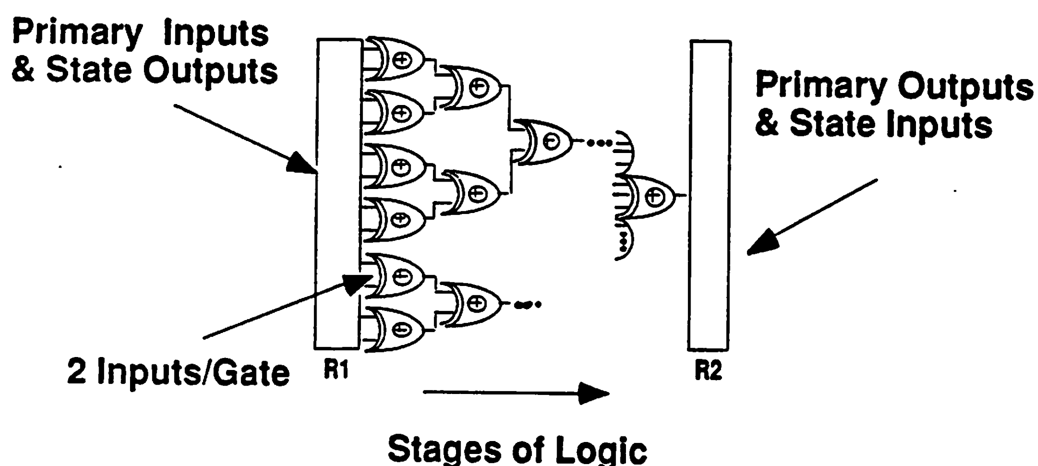

Figure 6.1: *100,000-input Parity Network: Two Inputs per Gate*

6.2 Synthesis Framework

The Multilevel Logic Interactive Synthesis System (MIS) [BR+87] is both an interactive and a batch-oriented multilevel logic synthesis and minimization system. The system starts from the combinational logic, generally extracted from a higher level description. It produces a multilevel set of optimized logic equations preserving the input-output behavior of the original description. The multilevel logic function is represented by a Boolean network. Each node in the Boolean network is a completely-specified Boolean function represented by both a sum-of-products form and a factored form. MIS is organized as a set of operators which are applied to the underlying Boolean network data structure. MIS has evolved as better algorithms are developed and implemented in the framework. The more recent evolution, MISII was used as the development framework for this

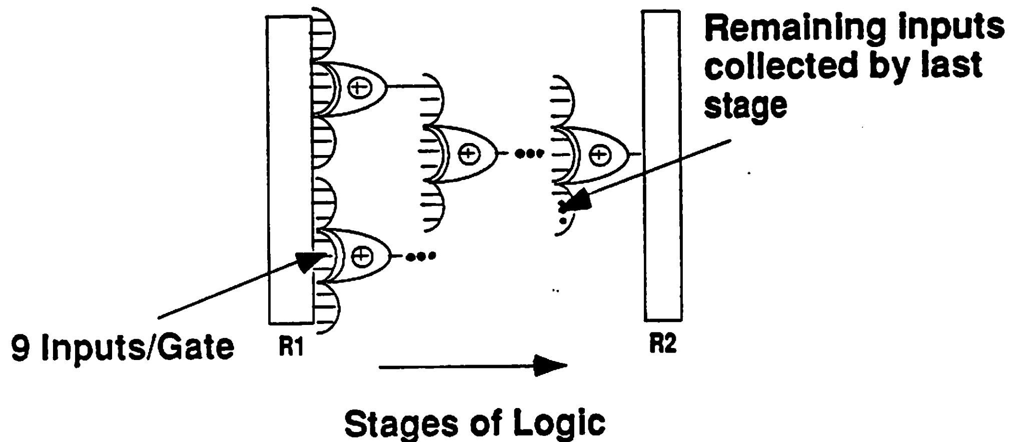

Figure 6.2: *100,000-input Parity Network: Nine Inputs per Gate*

report.

A common set of synthesis and optimization tools are available in the MIS framework for manipulation of the logic network, and the network can be mapped to the appropriate form for simulation. As an evolving system for development of new algorithms, the necessary programming interface is also available to allow the implementation of auxiliary simulation routines requiring access to the logic network.

6.3 Synthesis Tools

The MISII logic synthesis system allows access to some of the latest tools for logic optimization. The system contains tools for both local and global optimization of the network. The algorithms include network and node simplification and timing optimization, as well as tools for logic partitioning. Specific categories of these tools, which were studied in this project, are described in the following sections.

6.3.1 Conversion to Simulation Model: Mapping vs. Decomposition

In the context of this project, technology mapping and decomposition are tools for converting the circuit to a format for simulation on the massively parallel simulator. It is not expected to produce a reduction of the circuit, but to minimize the explosion of the circuit size, measured in

terms of the number of levels and gates, during the transformation for simulation.

The technology decomposition tool in MIS [BR+87], tech_decomp, performs a simple expansion of the logic function at each network node. The function is converted to an equivalent set of AND-OR gates. The internal representation of the node logic function in MIS is a two-level sum-of-products format, so the conversion is direct and efficient.

The technology mapping routines in MIS [DG+87] are intended to convert the technology independent MIS internal network representation to an equivalent logic circuit implementation in a specific technology. The technology is represented by a library of gates with predefined characteristics. These characteristics represent the physical behavior for the equivalent functional blocks implemented in the desired technology. The library information available to the technology mapping routines includes the physical size of the component layout and the actual delay times for signals through the component.

The mapper uses tree-matching algorithms to find valid combinations of library elements which are logically equivalent to the network. The library values of area and delay for these combinations are then calculated and the implementation chosen based on user-specified parameters for area and delay tradeoff.

For the purposes of massively parallel simulation, the area of the gate is irrelevant. The differences in delay between different functional elements of the library is also irrelevant, due to the minimal times required for function computation during the simulation. More important for simulation is the amount of input data required by the gate. Therefore, the only parameters of interest in the library are the delay values, which can be used to model the differences in communication required for simulation of gates with larger numbers of inputs.

A library can be constructed for mapping to a network that can be simulated by setting all library element area values to unity, or any simple equivalency value. The delay values will also be equivalent for elements with the same number of inputs, and increase with the number of inputs. The actual delay values reflect the tradeoff in the simulation model between the number of gate inputs and the number of logic levels in the network. Since inputs can be fetched in parallel, the number of gate inputs has a smaller effect on the final simulator performance than the number of levels. The delay values must be set to favor reduction in the number of levels over reduction in the number of gate inputs. This may be accomplished by making the differences in delay due to the number of inputs a fractional part of the total library element delays. For example, the basic delay can be set at unity with an additional delay of 5% per gate input over two. Using this approach, distinct weighting is set for optimizing for the number of levels before the number of gate inputs.

The mapping routines can be run with full delay optimization but with area optimization turned off. This effectively eliminates interest in the total number of gates in the final implementation, which doesn't directly affect the performance of the massively parallel simulation, given sufficient memory and processing elements.

6.3.1.1 Mapping Results

A comparison of the technology mapping and the technology decomposition tools was made with the results presented in Table 6.1. In this table, a number circuits from the MCNC and ISCAS logic synthesis benchmark sets, as well as some examples created by the author are shown. A more complete description of these examples is included in Chapter 7. In each case, *lvls* represents the number of levels of logic in the design implementation generated for implementation. *Ipc* represents the number of interprocessor communication steps required for simulation on a massively parallel computer. The experiments were made using the library described above for the technology mapper and varying numbers of possible inputs for the decomposition. For the tech_decomp results, the column title indicates the number of inputs (2 in, 3 in, etc.) to the AND and OR gates in the decomposition. A few more clarifications on the actual operation of the tools should be made with these results. In the mapping library, the functional complexity of the elements is limited by the function format. The tree-matching algorithms depend on the function designation of the element, so it is the library writer's job to include entries for all unique permutations of the function description. The efficiency of the mapper in finding improved implementations is directly dependent on the completeness of the library. The decomposition of nodes is restricted to AND and OR gates, with inversion of inputs allowed. As previously described, the decomposition is a direct conversion of a nodes logic function from the internal sum-of-products form, as long as the number of gate inputs specified to the tech_decomp routine is greater than the number of inputs to any single network node. Otherwise, the internal node is broken into a representation with smaller numbers of gate inputs, requiring additional gates and levels to represent the same logic function. Therefore, the number of gate inputs for the decomposition must be carefully chosen so that the number of gates and levels is not increased.

6.3.1.2 Mapping Conclusions

The technology decomposition routine produces almost ideal results, when the target simulation model is restricted to AND, OR, and INV functions. It results in small expansion in the

	map (best)		decomp (2 in)		decomp (3 in)		decomp (4 in)		decomp (5 in)		decomp (10 in)	
circuit	lvls	ipc	lvls	ipc	lvls	ipc	lvls	ipc	lvls	ipc	lvls	ipc
5xp1-hdl	14	28	19	38	14	38	14	43	14	43	14	43
C1355	22	49	25	50	24	49	22	49	22	51	22	51
C6288	120	240	120	240	120	240	120	240	120	240	120	240
des	33	66	30	60	23	68	19	71	19	71	18	126
duke2	18	36	9	18	6	18	5	20	4	20	4	36
misex3	17	34	13	26	10	30	8	29	7	30	5	36
sao2-hdl	36	72	36	72	31	79	27	91	22	82	22	91
seq	24	48	12	24	8	24	7	28	5	25	5	50
trisc	-	-	37	74	36	84	36	91	35	90	35	90
add16	31	62	46	92	32	94	31	107	31	107	31	107
add32	63	126	94	188	64	190	63	219	63	219	63	219
mult16	73	146	96	192	68	198	65	223	65	223	65	223
mult24	106	212	146	292	101	298	98	341	98	341	98	341
mult32	138	276	194	388	133	394	130	454	130	454	130	454

Table 6.1: Comparison of Technology Mapping and Decomposition.

number of levels, while producing the implementation for the massively parallel simulation. The functional restriction allows easy comparison of implementations, with minor functional sacrifice, considering the realistic restriction of the library functional descriptions. The technology decomposition is efficient, allowing quick comparison of implementations with varied maximum gate inputs. The efficiency makes it a natural choice for benchmarking runs. In practice, it also gives very good results compared to technology mapping using a technology library modified for massively parallel simulation.

6.3.2 Simplification Algorithms

Simplification techniques for multilevel logic networks are primarily concerned with optimization of individual node functions and some incremental improvements to the network structure. One of the most powerful techniques for node optimization is the use of two-level logic minimizers. The implicit don't care information available at a each node of a logic network is used to perform two-level logic minimization on the Boolean function associated with the node. The input to a two-level minimizer, such as ESPRESSO [B^+84], is composed of an onset cover and a don't care set. The onset cover is the node function expressed in terms of its inputs. The don't care set at each node contains information on the structure of the network. This information is a combination of external, observability, and satisfiability don't cares. Unfortunately, the don't care

set is extremely large, and may be restricted by the two-level minimizer.

For the basic simplify operation in the MIS synthesis system, ESPRESSO is used for the two-level minimization, utilizing varying amounts of don't care information. The don't care set is constructed from a subset of the observability don't care set. The extent of the don't care information can be varied from none, to the fanin don't cares of all levels of transitive inputs to the node being minimized.

A newer simplification algorithm has been developed, [STB91], which computes almost the full local don't care set at each node. This method uses external don't care information more effectively, using newer data manipulation techniques and image computation methods to find the local don't care sets at each node. In order to handle the information more efficiently, binary decision diagrams (BDD's) are used to generate and manipulate the logic information. By providing a more complete don't care set to the two-level minimizer, a better simplification result is generated. This technique is implemented as the full_simplify command in the MIS synthesis system.

6.3.3 Timing Algorithms

The need to synthesize circuits meeting strenuous timing requirements has led to the development of algorithms for reducing the delays in circuits. A subset of these methods, reducing delay in combinational logic, is of particular interest in this report. This method, known as timing restructuring, attempts to restructure the circuit globally to have better timing properties. In the restructuring, the quality of a circuit is judged by the circuit structure, rather than the calculation of detailed timing information.

A standard method of timing optimization maps the network into 2-input NAND gates and inverters and then uses a unit delay model to estimate the delays in the network. With these delays, critical paths in the circuit are identified. The timing optimization routine then attempts to restructure the circuit along these paths to reduce the overall delay in the circuit. With the 2-input NAND gate and inverter form for the network, the unit delay model equates the circuit delay to the number of logic levels. When the circuit is restructured with a shorter delay along the critical path, the circuit has fewer levels of logic, which is the desired result for reduced simulation time on the massively parallel simulator.

6.3.3.1 Speed_up

A routine for timing restructuring of combinational circuits, speed_up, is provided in the MIS synthesis system [BR+87]. The input Boolean network is decomposed into 2-input NAND gates and inverters and weights are assigned to the nodes in the resulting Boolean network. The algorithm uses the node weights to identify the critical portions of the network, choosing a set of nodes which will reduce the delays on all the critical paths when sped up. Partial collapsing is performed on the critical path at each node in the critical set. The collapsed nodes are then decomposed again into an alternate 2-input NAND gate and inverter representation. The process continues iteratively until no more improvement can be made.

The MIS speed_up command allows one of three delay models to be specified: mapped, unit fanout, or unit. The mapped delay model computes the delay using the delay data in the technology library, as described above with the technology mapper. The unit fanout model is intended to capture a technology independent model, assigning a 1 unit delay to each gate and 0.2 units to each fanout stem. The unit delay model counts the level of the circuit as its delay. The speed_up command provides additional control over the timing estimation and restructuring operations. Options exist to determine the nodes considered critical, the amount of collapsing to be performed around the critical node set, and the area-delay tradeoff to be considered. For this project, the unit delay model and the pure timing (no area) mode are used to target the massively parallel simulation model. The other parameters are investigated for their effects on generating the best results for simulation

6.3.3.2 Collapse

For pure timing optimization, a simple way to improve the performance is to collapse a logic network into two levels of logic. Unfortunately, this obvious technique is not widely applicable. For a large class of circuits, including circuits for which simulation is most expensive, collapsing is impractical due to memory and computation costs.

6.3.3.2 Reduce_depth

While the cost of collapsing into two levels of logic is prohibitive for most circuits, part of the delay improvement can be obtained by collapsing a circuit partially at a smaller cost. Delay optimization of combinational logic using clustering of nodes and partial collapsing was investigated by Touati [TSB91] and implemented in the MIS synthesis system. This algorithm performs a partial

collapse of the circuit based on delay-driven clustering. The circuit may then be simplified using the simplification tools discussed in section 6.3.2.

The reduce_depth command performs a partial collapse of a network by first clustering nodes and collapsing each cluster into a single node. The clusters are formed to minimize the number of levels of nodes in the network after the collapsing of the clusters. The clustering algorithm does not take into account the complexity of the logic functions in the nodes. Typically the input network is already decomposed into simple gates, but this is not enforced. The desired depth of the network after clustering may be specified, as well as the maximum amount of logic duplication allowed to obtain the clustering.

To use the reduce_depth clustering algorithms to target massively parallel simulation performance, the desired number of node levels is small, with no limits set on the amount of duplication. The node levels directly correspond to the logic levels in the simulation, while the duplication correlates to increased logic parallelism. Since the complexity of the clustered and collapsed nodes is not restricted, the resultant logic provides an opportunity for logic simplification. The use of full_simplify to optimize each node in the partially collapsed network should produce some logic reduction in the simulation. To ensure that the number of logic levels is not increased during the simplification phase, Touati added this restriction in determining acceptable simplifications. This optimization mode is used when the simplification step follows the reduce_depth execution.

Chapter 7

Experimental Results

7.1 Benchmark Circuits

Experiments were run on most of the circuits from the MCNC [Lis88] and ISCAS [BBK89] logic synthesis benchmark sets. Additional circuits were generated using the BLIS behavioral level synthesis system [Whi92]. The number of useful examples was restricted by the capacities of the synthesis tools. Specifically, a number of the circuits caused the synthesis tools to run out of memory, or failed to complete after a large amount of CPU time (weeks of real time). The *TRISC* example circuit is a full RISC microprocessor design which has been prepared for fabrication. The *par32* circuit is a 32 bit parallel arithmetic circuit, and the *MULT* circuits are multiplier circuits implemented using a Wallace Tree arrangement [Hwa79]. Each of these examples was completely generated from a behavioral-level description using the BLIS behavioral synthesis system. The experiments run on the circuits are characterized and the results obtained are presented in the following sections.

7.2 Performance Metrics

The simulation system described in this report must be evaluated on both the performance of the massively parallel machine for simulation and the effectiveness of logic synthesis tools to target this simulation platform. Dynamic performance of the massively parallel simulator is obtained by direct measurements of the simulation times on the benchmark circuits. The structure of the circuits predicts this performance directly. Therefore, the effectiveness of synthesis is measured by changes to the static structure of the circuits generated for simulation.

7.2.1 Dynamic Metrics

Dynamic performance of the simulation are affected by the static configuration of the network implementation. There are two major dynamic characteristics of SIMD simulation which must be addressed: computation and communication. The time required for the computation of the model function, i.e. gate evaluation, is minimized in this study by restricting the simulation model to simple logic functions. This translates directly to simple Boolean operations on the processor with minimum variation in evaluation time between possible gate types.

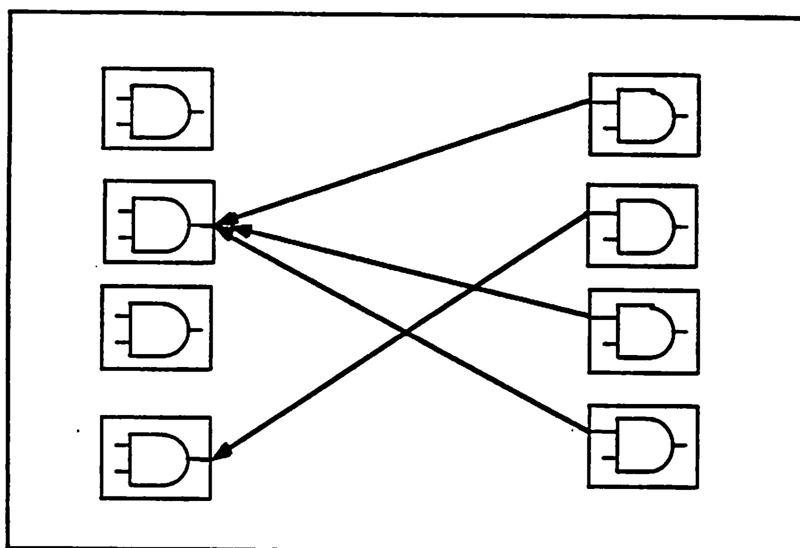

Figure 7.1: *Contention on a Single Input Fetch*

The second dynamic characteristic, communication time, is directly correlated to the static configuration of the network. Since the communication of data between processing elements is accomplished using the global routing network, the communication characteristic translates to a dependence on contention for the routing channels. Contention during data transfers can be caused by two processors trying to access data from the same source processor. An example of multiple fanouts from a single gate being updated in the same communication step is shown in Figure 7.1. This is a product of the static (fanout) structure of the network. Due to the design of the communication network on the MasPar computer, contention can also occur when two processors access physically close processing elements. This is the physical location in the processor array, which is set by the mapping of the network to the SIMD machine.

Tests were run to determine the effects of contention on the run-time performance of

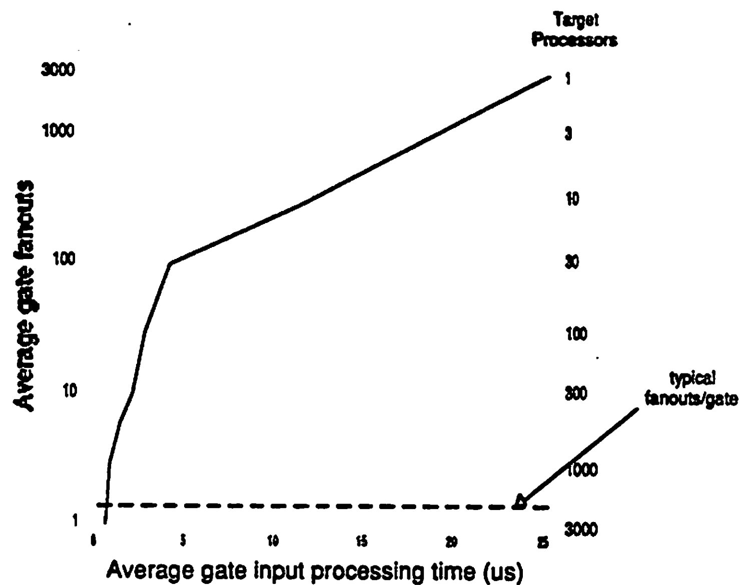

Figure 7.2: *Contention Time Comparison*

the simulation on the MasPar computer (see Figure 7.2). These tests measured the interprocessor communication times with 3000 processors and varying levels of contention. The time for the data transfer between processors ranges from 0.5 to 25 milliseconds, as the amount of contention increases from the no contention to all 3000 processors in contention. For the typical value of three fanouts per gate, the time is only increased by 0.2 ms. Therefore, the potential for large contention delays exist, but it is not expected to be a major factor in the simulation performance.

7.2.2 Static Metrics

Static characteristics of the logic networks directly correspond to the absolute simulation performance. These relationships are due to the simulation algorithm chosen and easily seen from the evaluation loop shown in Figure 5.1 (see Chapter 5). The main static characteristic affecting the evaluation time is the number of logic levels in the circuit implementation. The maximum number of inputs over all the logic gates at each logic level is the next most important characteristic. The number of input fetches to evaluate the gates at each level is set by the maximum number of inputs for any gate at that level, since the values can be transferred in parallel. The sum of these input fetches over all logic levels is the total number of interprocessor communication (ipc) steps required for simulation of a single input pattern. For example, consider a logic network consisting of N levels of two input nand gates. The maximum number of inputs at each level is 2, therefore the

number of ipc steps required is $2N$. Now replace a single gate in the first level of logic with a five input nand gate. The maximum number of inputs for the first level is 5 and for the other N-1 levels is 2. Therefore $2(N-1) + 5$ ipc steps are now required.

7.3 Results

The results for this project are presented in the following two sections, which correspond to the two concepts investigated. The first section presents the absolute performance of the simulator on a massively parallel machine. The simulation performance is measured on the original benchmark circuits before optimization with the synthesis tools. The second section details the effectiveness of logic synthesis tools for targeting the massively parallel simulation.

7.3.1 Simulation Performance

For the simple simulator implementation described, the results shown in Table 7.1 were collected. For each example, the table shows the size of the simulation input vector (In), the size of the simulation output vector (Out). The number of gates (gates) and the number of logic levels (lvls) in the synthesized circuit indicate the size of the logic networks. The number of communication steps (ipc) required for the simulation is twice the number of logic levels, since these circuits were all decomposed into two input logic gates. The circuits were simulated for a large number of random input patterns. The performance times (ms/pat) given are the calculated average time, in milliseconds, required to simulate a single input pattern. The final column (ms/ipc) is the time required for each interprocessor communication step in the simulation run. This value is obtained by dividing the time per pattern (ms/pat) by the number of communication steps required to simulate each pattern (ipc).

The absolute performance time of interest is the number of input patterns simulated per second. Looking at the ms/pat times for the benchmarks, note that there are only 5-20 input patterns simulated per second. This is at least an order of magnitude slower than expected (based on a rule of thumb performance of 1,000,000 gates / second for a sequential simulator). Since the performance bottleneck on this SIMD machine is in the interprocessor communication times tests were run to measure the base communication times of the MasPar machine. The times, for varying amounts of processor activity, ranged from 0.3 - 0.6 ms per interprocessor communication step (ipc). These times are consistent with the times used per ipc (ms/ipc) in the benchmarks simulation runs. Given

Circuit	In	Out	gates	lvls	ipc	ms/pat	ms/ipc
des	256	245	4,441	43	86	215.0	2.50
duke2	22	29	423	19	38	53.2	1.40
misex3	14	14	713	45	90	125.0	1.40
seq	42	35	1,964	26	52	124.0	2.40
trisc	1,176	832	5,546	63	126	185.9	1.48
par32	64	64	12,339	223	446	204.2	0.45
mult16	32	32	5,735	102	204	84.9	0.42
mult24	48	48	13,741	151	302	145.2	0.48
mult32	64	64	24,643	194	388	238.4	0.61

Table 7.1: Performance Results for the MP Simulator.

this machine communication time and the number of communication steps required for each circuit, this is the best performance expected from the simulator.

7.3.1.1 TRISC: An Example

For more insight into the simulator performance, the TRISC circuit is examined. This example produced simulation results for only 5 input patterns per second. To understand this performance, the effect of the circuit implementation on the simulation times was investigated. The number of logic gates at each logic level in the TRISC implementation is plotted Figure 7.3. Since the implementation was mapped using 2-input gates, this results in the processor activity at each communication step (ipc) shown in Figure 7.4. As determined by the number of logic gates and the number of logic levels, the average number of active processors is 88. The activity plots for some of the benchmark circuits is shown in Figure 7.5. These plots reflect the similar lack of utilization of resources for those simulations.

7.3.2 Synthesis Improvement to Simulation

The effectiveness of using synthesis tools for speeding up simulation is measured by static performance metrics. The results for each synthesis tool are, therefore, given in terms of their effect on the number of levels of logic in the implementation and the total number of interprocessor communication steps required for the simulation. These measurements depend on the simulation implementation previously described in Chapter 5.

To make the comparisons, the procedure described in Figure 7.6 was followed. For the

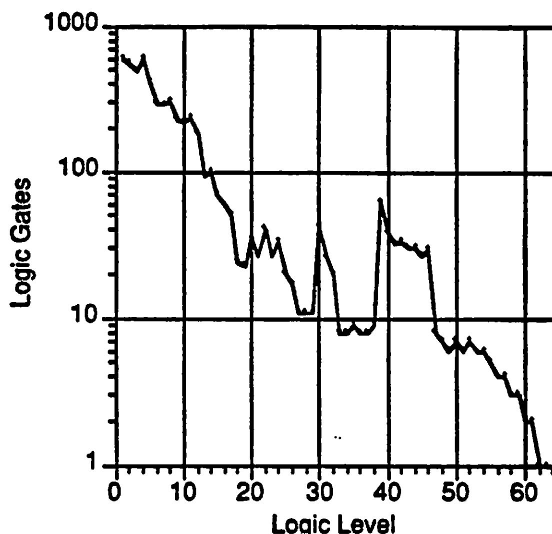

Figure 7.3: *TRISC Gates* vs. *Logic Level*

original circuit implementation, the synthesis / optimization step is skipped. The and-or-invert decomposition (Decompose to Simple Gates) step is run for varying numbers of gate inputs, with the implementation producing the fewest interprocessor communication steps chosen. The implementation numbers are calculated after a sweep operation to collapse all the inverters into their fanouts. This operation is valid for all simulation implementations since the simulation calculation was designed to handle inversion of the gate inputs. Finally, the simulation data is generated for the actual massively parallel simulation.

The results are presented below comparing the static metrics for the original circuit and alternate implementations generated. The tools used and the parameters specified for these tools are described with the results.

7.3.2.1 Simplify

Simplify performs local logic optimization on each node in the logic circuit. It is strictly targeted to reduce the complexity of the logic functions represented at these local data points. This may produce a circuit implementation that simulates faster. This is a result of reducing the logic complexity and is not due to any global timing considerations. The simplify option was used to restrict increases in the number of levels of logic by the simplification. This reduces the general effectiveness of the simplification algorithm but produces results targeting reduced simulation communication. The results of running simplify on the benchmark circuits are shown in Table 7.2.

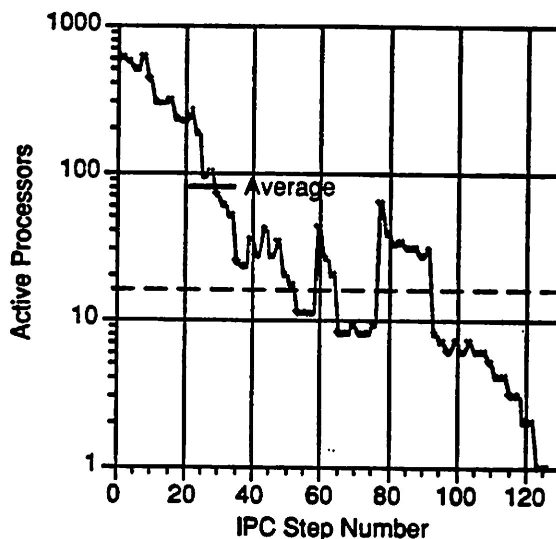

Figure 7.4: *TRISC Active Processors* vs. *IPC Step*

The simplify operation produced small reductions in the total number of gates for a few of the circuits. The number of communication steps required for simulation (ipc) was only reduced for one circuit, sao2-hdl, and only by 6%.

The full_simplify operation was executed using the same level restriction option with the results for the benchmark circuits shown in Table 7.3. The full_simplify operation could not be applied to many of the larger benchmark circuits due to memory limitations. The size of the circuits (gates) was reduced for most of the circuits. The number of communication steps required for simulation (ipc) was only reduced for two of the circuits, 5xp1-hdl and sao2-hdl, by 10% and 25%, respectively.

This simplification technique shows a possibility of improving the simulation performance by reducing the number of logic levels. The main result of the simplification, however, is fewer gates. This is likely to reduce the amount of parallelism in the circuit. The tool might be used in conjunction with other tools, which directly target the number of logic levels to provide consistently better results for simulation.

7.3.2.2 Collapse

The collapse of a circuit into two logic levels is ideal for explicitly reducing the number of logic levels in the circuit. The resulting circuit must still be decomposed into simple gates for the simulation. The decomposition into restricted input gates may cause a subsequent increase

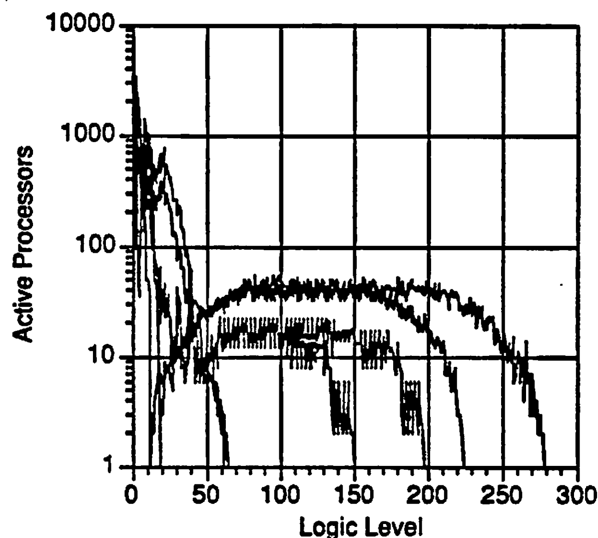

Figure 7.5: *Benchmark Processor Utilization*

in the number of logic levels to be simulated. Results are given in Table 7.4 for the collapsed two-level circuits and the same circuits after mapping to simple two input gates. The collapsed circuits have only two levels of logic, but have dramatic *increases* in simulation communication time (ipc). The collapse operation produced gates with large numbers of inputs. A high communication cost is incurred by the serial transfer of data to a single processing element (gate). Decreased communication was achieved by mapping to simple gates (and, or) restricted to two inputs. Some of the mapped circuits had significant improvement in simulation communication over the original circuits but others had no improvement. The main disadvantage to the collapse operation is the limited size of the circuits which may be collapsed. Exponential growth of the circuit during collapsing restricts the procedure to small or simple circuits which are expected to require less time for serial simulation.

7.3.2.3 Speed_up

The speed_up synthesis tool performs timing optimization on a circuit. Delays in the circuit are estimated, using a delay model, and critical paths are identified. Logic nodes along the critical paths are collapsed and then re-synthesized in an attempt to reduce the delay along these paths. Restricted to the unit delay model and pure timing optimization, speed_up minimizes the number of logic levels in the critical paths. With these restrictions, the results of several experiments with speed_up are shown in Table 7.5. The three experiments correspond to successively increasing

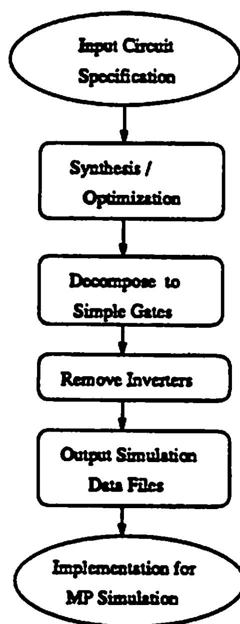

Figure 7.6: *Synthesis Optimization for MP Simulation*

the number of paths considered critical. For each of these experimental values, speed_up iterates over successively increased amounts of collapsing along the identified critical paths. The best of all these iteration results is returned. The results are presented for the circuit after mapping to simple input gates for massively parallel simulation. The number of gates in the simulation implementation (gates) provides a measure of the amount of circuit modification compared to the original circuit (orig), similarly mapped to simple gates.

Simulation improvements in terms of communication step reduction ranges from 0 - 75%. The largest improvements are in the reduction of the chain structure in the adder benchmark circuits. Speed_up is well-suited to optimizations for the massively parallel simulation algorithm, when modeled correctly as a pure timing optimization. The drawback of speed_up is the isolation of optimization to the critical paths, without exploring the possibility of more global optimizations.

7.3.2.4 Reduce_depth

The second timing optimization algorithm, reduce_depth, is also directly applicable to the problem of simulation performance on parallel machines. Similar to speed_up it attempts to reduce the total delay of the circuit. Restricted to a unit delay model, reduce_depth performs operations to reduce the number of logic levels (lvls) and consequently, the number of communication steps (ipc) required for circuit simulation on a massively parallel machine. Reduce_depth

	orig			simplify		
circuit	lvls	ipc	gates	lvls	ipc	gates
5xp1-hdl	19	38	93	19	38	93
C1355	25	50	518	25	50	518
C6288	120	240	2353	120	240	2353
des	30	60	3303	30	60	3298
duke2	9	18	1717	9	18	1717
misex3	13	26	1590	13	26	1560
sao2-hdl	36	72	326	34	68	326
seq	12	24	17788	12	24	17152
trisc	37	74	4472	37	74	4472
add16	46	92	244	46	92	244
add32	94	188	500	94	188	500
mult16	96	192	4107	96	192	4107
mult24	146	292	9751	146	292	9751
mult32	194	388	17623	194	388	17623

Table 7.2: Results for **simplify**.

includes logic node clustering and duplication techniques which trade circuit area for improvements in delay. This corresponds directly to increasing the amount of parallel execution to reduce the serial execution in parallel simulation. This is identical to the goal of synthesis for simulation on massively parallel machines.

The results of the reduce_depth operation on the benchmark circuits are given in Table 7.6. The reduction in the number of communication steps (ipc) ranges from 0 - 48%. The amount of duplication of gates traded for the communication reductions can be compared by the number of simple gates (gates) in the simulation decomposition. Additional data is supplied in Table 7.7 when a logic simplification algorithm is used with the reduce_depth command. The full_simplify algorithm described in Chapter 6 is applied to the collapsed node clusters, prior to mapping them for simulation. This produces incremental improvement over the reduce_depth technique used alone.

	orig			full_simplify		
circuit	lvls	ipc	gates	lvls	ipc	gates
5xp1-hdl	19	38	93	17	34	85
C1355	25	50	518	25	50	510
C6288	120	240	2353	-	-	-
des	30	60	3303	30	60	3223
duke2	9	18	1717	9	18	1717
misex3	13	26	1590	13	26	1455
sao2-hdl	36	72	326	27	54	202
seq	12	24	17788	12	24	17152
trisc	37	74	4472	37	74	4449
add16	46	92	244	46	92	244
add32	94	188	500	94	188	500
mult16	96	192	4107	-	-	-
mult24	146	292	9751	-	-	-
mult32	194	388	17623	-	-	-

Table 7.3: Results for **full_simplify**.

	orig			collapse			collapse (2 in)		
circuit	lvls	ipc	gates	lvls	ipc	gates	lvls	ipc	gates
5xp1-hdl	19	38	93	2	25	85	8	16	290
C1355	25	50	518	-	-	-	-	-	-
C6288	120	240	2353	-	-	-	-	-	-
des	30	60	3303	-	-	-	-	-	-
duke2	9	18	1717	2	31	29	9	18	1717
misex3	13	26	1590	2	204	1491	12	24	13902
sao2-hdl	36	72	326	2	38	86	9	18	521
seq	12	24	17788	2	129	1491	12	24	17788
trisc	37	74	4472	-	-	-	-	-	-
add16	46	92	244	-	-	-	-	-	-
add32	94	188	500	-	-	-	-	-	-
mult16	96	192	4107	-	-	-	-	-	-
mult24	146	292	9751	-	-	-	-	-	-
mult32	194	388	17623	-	-	-	-	-	-

Table 7.4: Results for **collapse**.

	orig			speed_up (t=0.5)			speed_up (t=1.0)			speed_up (t=2.0)		
circuit	lvls	ipc	gates	lvls	ipc	gates	lvls	ipc	gates	lvls	ipc	gates
5xp1-hdl	19	38	93	8	16	147	9	18	120	10	20	119
C1355	25	50	518	17	34	612	20	40	532	20	40	532
C6288	120	240	2353	100	200	2894	119	238	2358	104	208	2454
des	30	60	3303	18	36	3506	19	38	3431	20	40	3398
duke2	9	18	1717	10	20	593	10	20	596	10	20	602
misex3	13	26	1590	13	26	724	13	26	724	13	26	724
sao2-hdl	36	72	326	30	60	211	30	60	225	29	58	553
seq	12	24	17788	14	28	2319	14	28	2383	15	30	2513
trisc	37	74	4472	24	48	4654	25	50	4548	26	52	4520
add16	46	92	244	13	26	312	12	24	296	13	26	346
add32	94	188	500	21	42	564	21	42	581	20	40	688
mult16	96	192	4107	57	114	3256	51	102	3319	49	98	3334
mult24	146	292	9751	90	180	7715	96	192	7657	93	186	7698
mult32	194	388	17623	122	244	13764	128	256	13704	128	256	13710

Table 7.5: Results for **speed_up**.

	orig			reduce_depth		
circuit	lvls	ipc	gates	lvls	ipc	gates
5xp1-hdl	19	38	93	8	16	290
C1355	25	50	518	22	44	26304
C6288	120	240	2353	78	156	5498
des	30	60	3303	-	-	-
duke2	9	18	1717	9	18	1717
misex3	13	26	1590	13	26	2416
sao2-hdl	36	72	326	24	48	1749
seq	12	24	17788	12	24	17788
trisc	37	74	4472	30	60	31197
add16	46	92	244	25	50	33646
add32	94	188	500	49	98	77012
mult16	96	192	4107	-	-	-
mult24	146	292	9751	-	-	-
mult32	194	388	17623	-	-	-

Table 7.6: Results for **reduce_depth**.

	orig			reduce_depth		
circuit	lvls	ipc	gates	lvls	ipc	gates
5xp1-hdl	19	38	93	8	16	290
C1355	25	50	518	16	32	912
C6288	120	240	2353	82	164	5498
des	30	60	3303	-	-	-
duke2	9	18	1717	9	18	1717
misex3	13	26	1590	13	26	2253
sao2-hdl	36	72	326	19	38	463
seq	12	24	17788	12	24	17277
trisc	37	74	4472	29	58	17244
add16	46	92	244	25	50	33646
add32	94	188	500	49	98	77012
mult16	96	192	4107	-	-	-
mult24	146	292	9751	-	-	-
mult32	194	388	17623	-	-	-

Table 7.7: Results for **reduce_depth** with simplification.

Chapter 8

Conclusions and Future Work

In this project synthesis tools were used to map digital circuit descriptions to a parallel simulation format for evaluation on a massively parallel SIMD machine. Due to the inherent random structure of the gate-level logic descriptions being simulated, the local communication mechanism of the parallel machine could not be exploited effectively. As noted in Chapter 5, all communications between processors is via the much slower global communication routing mechanism. The measurement of the communication performance in Chapter 7 shows that the massively parallel SIMD global communication is too slow to be viable as a basis for a simulation engine. An order of magnitude improvement in the global communication performance must be obtained to provide feasible simulation performance for such circuits.

Moderate success was achieved using existing synthesis tools to target simulation performance rather than final chip performance. The success was made using timing optimization (delay reduction) tools. The final goal of these tools is similar to the optimization target for massively parallel simulation. By modifying the execution parameters of the tools, circuit implementations were generated requiring less interprocessor communication for simulation. The synthesis improvements are overwhelmed by the slow speed of the actual simulator on the massively parallel SIMD machine. While the use of SIMD machines does not appear promising, advances in the communication efficiency may occur to make this approach feasible.

The use of synthesis tools for other target machines should be investigated further. Moreover, new synthesis and optimization algorithms should be developed to optimize broader performance metrics of simulation. The possibility of different classes of high performance simulation engines provides the potential for simulation optimization in larger areas of design.

Bibliography

[B+84] R. K. Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Press, 1984.

[B+87] Z. Barzilai et al. HSS - a high-speed simulator. *IEEE Transactions on Computer-Aided Design*, 6(4), July 1987.

[BBK89] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of the sequential benchmark circuits. In *Proc. of the 1989 International Symposium on Circuits and Systems*, 1989.

[Bla90] Tom Blank. The MasPar MP-1 architecture. In *Proc. of the IEEE Compcon Spring 1990*, February 1990.

[BR+87] R. K. Brayton, R. Rudell, et al. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, 6(5), November 1987.

[Bri89] Jack. V Briner. LDVSIM: A mixed level system simulator. Technical Report 89-29, Duke University, Department of Computer Science, Durham, North Carolina, 1989.

[Bry84] Randal E. Bryant. A switch-level model and simulator for MOS digital systems. *Transactions on Computers*, C-33(2), February 1984.

[Bry87] Randal E. Bryant. COSMOS: A compiled simulator for MOS circuits. In *Proc. of the 24th Design Automation Conference*, 1987.

[Bry88] Randal E. Bryant. Data parallel switch-level simulation. In *Proc. of the 25th Design Automation Conference*, 1988.

[BS88] Mary L. Bailey and Lawrence Snyder. An empirical study of on-chip parallelism. In *Proc. of the 25th Design Automation Conference*, 1988.

[CE75] R. D. Chamberlain and M. N. Edelman. Lsim2 user's manual. Technical Report WUCS-88-3, Washington University, Department of Computer Science, St. Louis, Missouri, 1975.

[DG+87] E. Detjens, G. Ganot, et al. Technology mapping in MIS. In *Proc. of the International Conference on Computer Aided-Design*, 1987.

[Erd89] D. Erdman. *The Newton Waveform Relaxation Approach to the Solution of Differential Algebraic Systems for Circuit Simulation.* PhD thesis, Duke University, Durham, North Carolina, 1989.

[Fra86] E. Frank. Exploiting parallelism in a switch level simulation machine. In *Proc. of the 13th Int'l Symposium on Computer Architecture*, 1986.

[Gen85] GenRad Fareham Design Automation Products, Waterside Gardens, Fareham, England. *HILO-3 User's Manual*, 1985.

[Hil86] W. D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, Mass, 1986.

[Hwa79] Kai Hwang. *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley & Sons, New York, 1979.

[KBR89] Saul A. Kravitz, Randal E. Bryant, and Rob A. Rutenbar. Massively parallel swtich-level simulation: A feasibility study. In *Proc. of the 26th Design Automation Conference*, 1989.

[Lis88] R. Lisanke. Logic synthesis and optimization benchmarks user guide version 2.0. Technical report, MCNC, P.O. Box 12889, Research Triangle Park, N.C. 27709, December 1988.

[Nag75] L. W. Nagle. SPICE2: A computer program to simulate semiconductor circuits. Technical Report ERL-M520, University of California, Berkeley, May 1975.

[New79] A. R. Newton. The simulation of large scale integrated circuits. *IEEE Transactions on Circuits and Systems*, September 1979.

[Nic90] John Nickolls. The design of the MasPar MP-1: A cost effective massively parallel computer. In *Proc. of the IEEE Compcon Spring 1990*, February 1990.

[NSV83] A. R. Newton and A. L. Sangiovanni-Vincentelli. Relaxation based circuit simulation. *IEEE Transactions on Electronic Devices*, 30(9), September 1983.

[SB87] Larry Soule and Tom Blank. Statistics for parallelism and abstraction level in digital simulation. In *Proc. of the 24th Design Automation Conference*, 1987.

[SB88] Larry Soule and Tom Blank. Parallel logic simulation on general purpose machines. In *Proc. of the 25th Design Automation Conference*, 1988.

[SD80] K. Sakallah and S. W. Director. An activity directed circuit simulation algorithm. In *Proc. of the IEEE International Conference on Circuits and Computers*, October 1980.

[STB91] Hamid Savoj, Herve J. Touati, and Robert K. Brayton. Extracting local don't cares for network optimization. In *Proc. of the International Conference on Computer Aided-Design*, 1991.

[Ter83] C. Terman. RSIM - a logic-level timing simulator. In *Proc. of the IEEE Conference on Computer Design*, 1983.

[TSB91] Herve J. Touati, Hamid Savoj, and Robert K. Brayton. Delay optimization of combinational logic circuits by clustering and partial collapsing. In *Proc. of the International Conference on Computer Aided-Design*, 1991.

[Vel90] Beverly L. Vellandi. *Parallelism Extraction and Program Restructuring for Parallel Simulation of Digital Systems*. PhD thesis, University of Colorado, Boulder, Colorado, 1990.

[Whi92] Gregory S. Whitcomb. *Synthesis of Control-Dominated Digital Systems*. PhD thesis, University of California, Berkeley, 1992.

[WHPZ87] L. Wang, N. Hoover, E. Porter, and J. Zasio. SSIM: A software levelized compiled-code simulator. In *Proc. of the 24th Design Automation Conference*, 1987.

[Won86] K. F. Wong. Statistics on logic simulation. In *Proc. of the 23rd Design Automation Conference*, 1986.

[WSV87] D. Webber and A. Sangiovanni-Vincentelli. Circuit simulation on the Connection Machine. In *Proc. of the 24th Design Automation Conference*, 1987.