A Parallel Processing Architecture for CAD of Integrated Circuits

Bruce A. Tonkin, B.E.(Hons.)

Department of Electrical and Electronic Engineering
The University of Adelaide
Adelaide, South Australia.

September 1990

awarded 30.11.90

Contents

Abstract x

Declaration xi

Acknowledgements xii

1 Introduction 1

1.1 Computer Aided Design of Integrated Circuits 1

1.2 Synthesis Tools 6

1.2.1 Algorithms 6

1.2.1.1 Heuristic Algorithms 7

1.2.1.2 Simulated Annealing 8

1.2.1.3 Knowledge-based Systems 10

1.2.2 Examples 10

1.2.2.1 Cell generators 10

1.2.2.2 Compactors 11

1.2.2.3 Routers 12

1.2.2.4 Placement tools 12

1.2.2.5 Silicon compilers 12

1.3 Static Analysis Tools 13

1.3.1 Algorithms 13

1.3.1.1 Layout analysis algorithms 13

1.3.1.2 Network analysis algorithms 14

1.3.1.3 Test generation algorithms 15
1.3.1.4 Hierarchical analysis 17
1.3.2 Examples 17
1.3.2.1 Circuit extractors 17
1.3.2.2 Network comparators 18
1.3.2.3 Design rule checkers 18
1.3.2.4 Electrical rule checkers 19
1.3.2.5 Timing verifiers 19
1.3.2.6 Test generation tools 20
1.4 Dynamic Analysis Tools 20
1.4.1 Algorithms 21
1.4.1.1 Direct methods 21
1.4.1.2 Relaxation methods 24
1.4.1.3 Event driven simulators 25
1.4.1.4 Device models 26
1.4.2 Examples 26
1.4.2.1 Circuit simulators 27
1.4.2.2 Timing simulators 27
1.4.2.3 Logic simulators 28
1.4.2.4 Fault simulation 29
1.4.2.5 Functional simulators 30
1.4.2.6 Behavioural simulators 31
1.4.2.7 Mixed level simulators 31
1.4.2.8 Mixed mode simulators 31
1.5 Summary 32

2 Background **35**

2.1 Real Time Subsystem (RTS) 36
2.1.1 Introduction 36
2.1.2 Real Time versus General Purpose Computer Systems 38

2.1.2.1 Real time computer systems 38
2.1.2.2 General purpose computer systems 38
2.1.3 Real Time System Overview 41
2.1.4 Conclusions 44
2.2 Parallel Circuit Extraction 48
2.2.1 Introduction 48
2.2.2 Overview of Circuit Extraction 50
2.2.3 Goalie 51
2.2.4 Strategy for Parallel Circuit Extraction 55
2.2.5 Parallel Goalie 58
2.2.5.1 General purpose multicomputer 58
2.2.5.2 Serial disk accesses 61
2.2.5.3 Parallel disk accesses 61
2.2.5.4 Circuit partitioning 64
2.2.5.5 Layout analysis 65
2.2.5.6 Merging region numbers 66
2.2.6 Results 70
2.2.7 Conclusions 77
2.2.8 Acknowledgements 81

3 Approaches to Accelerating Computer-Aided Design 82
3.1 Introduction 82
3.2 Strategies for High Performance 84
3.2.1 SISD 85
3.2.1.1 RISC versus CISC 85
3.2.2 SIMD 86
3.2.2.1 Processor arrays 87
3.2.2.2 Vector processors 88
3.2.2.3 Multi-operation 89
3.2.3 MIMD 91

3.2.4 Summary 92
3.3 Characteristics of MIMD architectures 92
3.3.1 Circuit Technology 93
3.3.2 Multiprocessor 95
3.3.3 Multicomputer 96
3.3.4 Grain Size 97
3.3.5 Type of Processing Node 99
3.3.6 Interconnection Network 100
3.4 Parallel processing for CAD 103
3.4.1 Synthesis Tools - Simulated Annealing 103
3.4.1.1 Summary 111
3.4.2 Static Analysis Tools 112
3.4.2.1 Design rule checking 112
3.4.2.2 Circuit extraction 115
3.4.2.3 Test generation 115
3.4.2.4 Summary 117
3.4.3 Dynamic Analysis Tools 119
3.4.3.1 Circuit simulation 119
3.4.3.2 Timing simulation 123
3.4.3.3 Logic simulation 124
3.4.3.4 Fault simulation 132
3.4.3.5 Summary 133
3.5 Conclusions 134

4 Computer architecture for VLSI CAD **137**
4.1 Introduction 137
4.2 Workstations 140
4.3 Computer Node Architecture 145
4.3.1 VLSI Microprocessor Technology 145
4.3.2 Floating Point Support 148

4.3.3 Virtual Memory Management 148
4.3.4 Local Memory 149
4.4 Interconnection Network 150
4.4.1 Nectar 151
4.4.2 Network Topology 156
4.4.3 Scalable Coherent Interconnect (SCI) 157
4.4.4 High Speed Channel (HSC) standard 161
4.4.5 Fiber Distributed Data Interface (FDDI) 162
4.4.6 Interconnection Network Requirements 163
4.5 Secondary Storage 166
4.5.1 Disk Arrays 167
4.5.2 Bridge File System 172
4.5.3 Secondary Storage Requirements 173
4.6 Operating System 177
4.6.1 Amoeba 178
4.6.2 Mach 181
4.6.3 Operating System Requirements 183
4.7 Parallel Programming 187
4.7.1 Message versus Shared Variable Programming Paradigms188
4.7.2 Shared variables in a Loosely Coupled Environment . . 190
4.7.3 Linda 190
4.7.4 Distributed Shared Virtual Memory 192
4.7.5 Shared Data Object Model 195
4.7.6 Computer-aided Programming 198
4.7.6.1 CAPER 199
4.7.6.2 Hypertool 200
4.7.7 Programming Requirements 201

5 Conclusions 206

A Real Time Subsystem Architecture **211**

A.1 Commercial Alternatives 211

A.2 Specification for the Real Time Subsystem 214

A.3 Architecture of the Real Time Subsystem 218

A.3.1 VMEbus Backplane 219

A.3.2 Central Processing Unit 220

A.3.3 On-board Software 223

A.3.4 A/D Converter Module 223

A.3.5 D/A Converter Module 225

A.3.6 Timer/Parallel Interface Module 227

A.3.7 Data Link 227

A.3.8 Software Support 231

A.3.9 Summary 231

A.3.10 Acknowledgements 232

Bibliography **233**

List of Figures

1.1 Design Hierarchy 2
1.2 Design Representation Hierarchy 3
1.3 Ideal Design Cycle 5
1.4 Simulated Annealing Algorithm 9
1.5 Direct Method Algorithm 23

2.1 System Overview 42
2.2 Real Time Subsystem (RTS) Architecture 43
2.3 Example of a region merge 54
2.4 Partitioning strategies 56
2.5 HPC Architecture 59
2.6 Programmer's Model of the HPC 60
2.7 Programming Model used in Parallel Goalie 63
2.8 Example of merging across slice boundaries 67
2.9 Example of merging across slice boundaries (ctd.) 68
2.10 Performance results for the detection and numbering of transistor regions 70
2.11 Partitioning for circuit 3 with 12 processors 73
2.12 Load variation for circuit 3 with 12 processors 73
2.13 Number of scanline stops for circuit 3 with 12 processors . . . 74
2.14 Partitioning for circuit 2 with 12 processors 75
2.15 Load variation for circuit 2 with 12 processors 75
2.16 Number of scanline stops for circuit 2 with 12 processors . . . 76

4.1 System Architecture 138
4.2 Leopard multiprocessor workstation [Ashenden et al., 1989] . . 142
4.3 Intel i860 microprocessor layout [Kohn & Margulis, 1989] . . . 147
4.4 Single HUB Cluster of Nectar network [Arnould et al., 1989] . 155
4.5 SCI applications 159
4.6 SCI Transaction phases [IEEE P1596, 1990] 160
4.7 HSC Interface signals [Chlamtac & Franta, 1990] 162
4.8 Interconnection system 164
4.9 Disk array 168
4.10 Interleaved parity sectors [Katz et al., 1989a] 170
4.11 Secondary storage architecture 176

A.1 MC500 Minicomputer Architecture [Masscomp,] 212
A.2 System Overview 215
A.3 Real Time Subsystem (RTS) Architecture 219
A.4 Data Link structure 229

List of Tables

2.1 Characteristics of the circuits in Figure 2.10 71

2.2 Times for the slowest node out of 12 nodes 72

Abstract

Designers use computer-aided-design (CAD) software to check that an integrated circuit will function correctly and meet performance specifications. The software must produce results quickly enough for a designer to interactively evaluate many design iterations before fabricating the circuit. For large circuits, present workstations are not fast enough to run the software at the rate needed by designers. This has lead to research into parallel processing techniques to reduce the running time of CAD software.

The goal of this thesis is to describe a general purpose parallel computing system that should be capable of supporting the full range of design tools, and offer a substantial speed up to the design verification step in the design cycle.

The thesis begins by discussing the computational characteristics of modern CAD software. It then discusses initial ideas for the hardware and software attributes of a multiple processor computer system suitable for running CAD software. These ideas derive from the author's experiences with the design and construction of a high speed processing system, which can acquire and generate analog signals in real time, and the implementation of a parallel circuit extraction algorithm on a multicomputer.

The thesis goes on to review different architectural approaches to designing faster computers. before recommending a Multiple Instruction stream / Multiple Data stream (MIMD) architecture because it offers the flexibility required by CAD algorithms. It then analyses recent research into parallel CAD algorithms for running on MIMD architectures. On the basis of this analysis, the thesis recommends using a heterogeneous multicomputer MIMD architecture.

The thesis concludes by discussing the hardware and software requirements of a multicomputer architecture suitable for developing and running CAD software. The conclusions come from an analysis of recent research on the following components of the multicomputer: the processing node architecture, the interconnection network, the secondary storage system. the operating system, and the programming environment.

Declaration

This thesis has been submitted to the Faculty of Engineering at the University of Adelaide for examination in respect of the Degree of Doctor of Philosophy.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University, and to the best of the author's knowledge and belief contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

The author hereby consents to this thesis being made available for photocopying and for loans as the University deems fitting, should the thesis be accepted for the award of the Degree.

Bruce Tonkin
September 1, 1990

Acknowledgements

I would like to acknowledge the financial assistance of a Priority Commonwealth Postgraduate Scholarship and an Australian Telecommunications and Electronics Research Board (ATERB) Scholarship. Without this assistance I would never have persevered with my PhD research.

I thank the VLSI Systems research department at AT&T Bell Laboratories, Holmdel, NJ, USA for providing the opportunity to get experience with programming an existing parallel machine. Thanks to Bob Gaglianello, Howard Katseff, Beth Robinson and Teri Lindstrom for their help and advice in using the HPC multicomputer. Thanks to the Maple Ave. group including Alex Dickinson, Mary Keefe, Richard Reed and Fiona MacDonald for providing a stimulating living environment while in the US. Special thanks to Bryan Ackland for arranging my year at the Labs, and providing encouragement and advice.

I thank the Association for Computing Machinery's (ACM) Special Interest Group on Design Automation (SIGDA) for providing travel support to allow me to present a paper at the 1990 Design Automation Conference in Florida.

I thank both the technical and administrative staff of the Department of Electrical and Electronic Engineering, particularly Norman Blockley and Mary Parry, for their day to day assistance. Special thanks to Michael Liebelt for taking the time to listen to my ideas and proof reading my thesis. Special thanks also to my supervisor Doug Pucknell for providing guidance and encouragement.

Finally, and most importantly, thank you to my parents for putting up with a perpetual student.

B. A. T.

Chapter 1

Introduction

This introduction will give an overview of the computer aided design tools currently in use, and will discuss the computational characteristics of the algorithms that these tools use.

1.1 Computer Aided Design of Integrated Circuits

Computer aided design shortens the time for a designer to design an integrated circuit that meets its specifications. The objectives of computer aided design are:

- to shorten the time for design synthesis,
- to minimize design mistakes,
- to aid design changes,
- and to shorten the time for design verification.

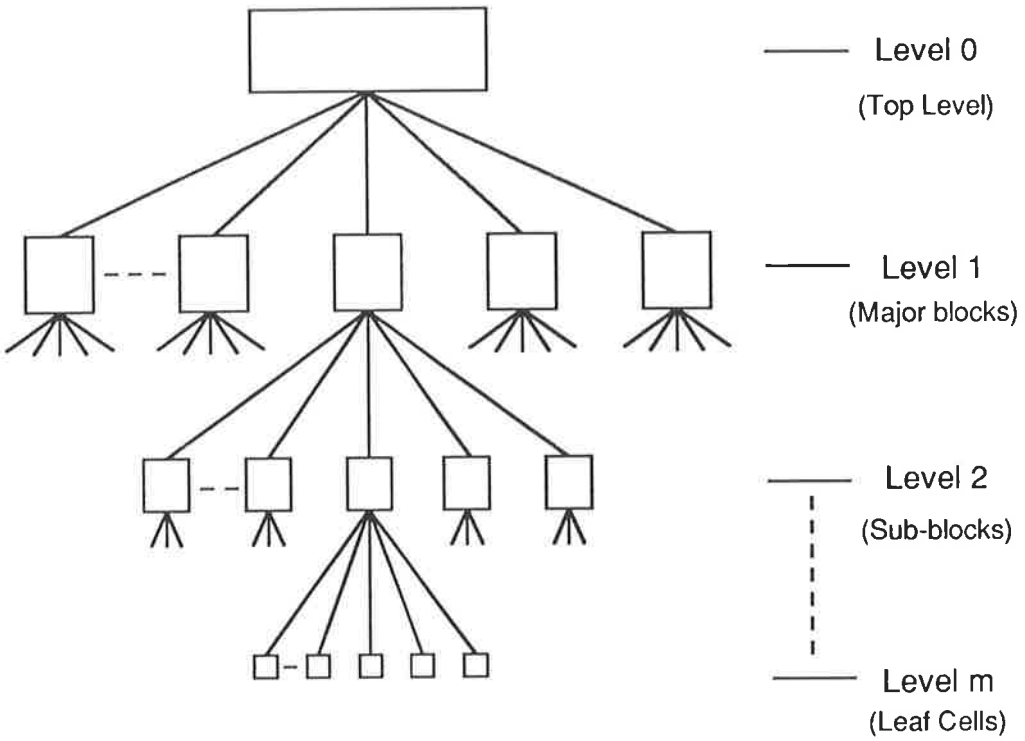


Figure 1.1: Design Hierarchy

The time required to design and develop integrated circuits is increasing rapidly as they become more complex. Computer aided design reduces this time making it economical to take advantage of improving technology, and to produce competitively priced products. It has made it possible to use integrated circuit (IC) technology in low production volume applications.

Usually a designer develops a design hierarchically to help minimize the design detail that needs to be considered at once. Figure 1.1 illustrates this hierarchy. The designer divides the top level of the design into several blocks, which are as self contained as possible, with few connections between them. They represent major functional units such as arithmetic units, memory units, and control units. For a large design, separate designers may develop these units. Designers may further subdivide the major blocks until they

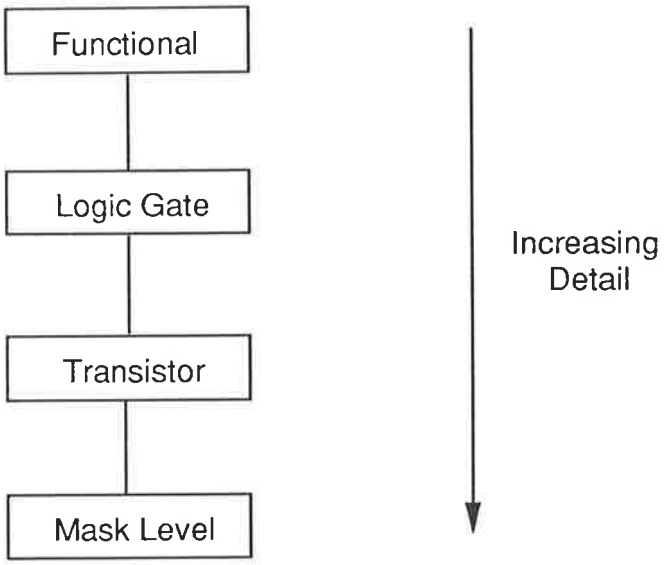


Figure 1.2: Design Representation Hierarchy

obtain a set of simple leaf cells. Typically each block consists of between 5 and 9 sub-blocks. The number of sub-blocks used is a trade off between too many objects for a designer to consider at once, and a hierarchy that is too deep. The leaf cells usually have less than 100 gates, and their design takes into account the analog characteristics of the chosen fabrication technology. Each level of design consists of one or more instances of any of the blocks at lower levels of the hierarchy. Standard cell libraries contain the blocks in the lower levels of the hierarchy that fabrication companies, or other designers, may have previously designed. Computer aided design software can automatically generate the information required for fabrication of the complete circuit, from the hierarchical description of the design.

Figure 1.2 shows a hierarchy of design representations available to describe each block of a design as its design progresses. A functional specification defines the behaviour of the block in response to its input signals. Logic gates describe a block of digital circuitry. The final result of the design

process is a description of the masks to be used in the fabrication process. The designer can use the computer to check each representation against the higher levels for consistency.

Design tools are the computer programs that aid the designer. Complete automation of the design task is unlikely to be achieved for the general case, because technology changes before tools can reliably produce an optimum design. Instead, computers relieve the designer from the repetitive and time consuming tasks of circuit design. The designer still needs to make the complex and critical decisions.

Design tools should allow the designer to remove the costly and time consuming process of circuit fabrication from the design cycle, as Figure 1.3 illustrates. Although, for high production volume applications, further refinements may improve the yield sufficiently to justify further design iterations after initial fabrication.

Rubin [Rubin, 1987] gives a good overview of the design tools available, and their application to circuit design. Case studies [Tredennick, 1987; Ditzel & Berenbaum, 1987; Bosshart, 1987; Bays et al., 1989] describe the use of some design tools in commercial designs. There are two main classes of design tools: synthesis tools and analysis tools.

Synthesis tools are used to aid the construction of a circuit. These include tools to automatically generate the design of specific blocks, tools to arrange the placement of sub-blocks within a block, and tools to route the interconnections between sub-blocks. They shorten the design time and allow design changes late in the development process. Preas and Lorenzetti [Preas & Lorenzetti, 1988], and Ohtsuki [Ohtsuki, 1986], bring together recent work by researchers in the development of these tools.

Analysis tools verify that the circuit construction is correct. They minimize the number of design mistakes, and they shorten the time required for design

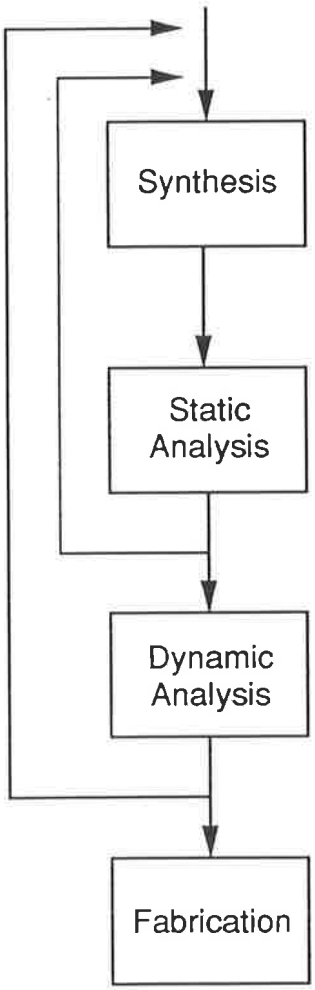


Figure 1.3: Ideal Design Cycle

verification. There are two broad classes of analysis: static analysis and dynamic analysis.

Static analysis tools analyse circuit properties that are independent of the electrical state of the circuit. These include tools to check design rules, tools to extract circuit parameters, and tools to generate test vectors.

Dynamic analysis tools analyse the operation of the circuit in response to input signals. These tools check that the representations of the circuit are functionally correct, and they check the performance characteristics of the circuit.

The following sections describe the algorithms used in synthesis tools, static analysis tools, and dynamic analysis tools, along with examples of their use.

1.2 Synthesis Tools

Synthesis tools try to automate the task of circuit layout. The goal is to optimize the design given a set of design constraints, such as the maximum power dissipation, the minimum speed of operation, and the maximum surface area.

1.2.1 Algorithms

Most of the tasks associated with automating circuit layout are *combinatorial optimization problems.* They each consist of a finite set of solutions along with a cost function. The cost function assigns a numerical value for each solution that relates its quality to the other solutions. The task of finding an optimal solution to the layout problem is NP-hard [Garey & Johnson, 1979]. This means that there is no known algorithm that guarantees to find the optimal solution in a time that increases polynomially with problem size. Because of the many possible solutions, it is impracticable to generate each

solution and evaluate its cost function to find the optimal solution. Instead, computers use *heuristic* algorithms to try to find a good solution that is not necessarily the optimum solution. Lengauer [Lengauer, 1988], and Shing and Hu [Shing & Hu, 1986], discuss the complexity of layout algorithms in more detail.

The software developer chooses the method to solve a particular layout problem by considering the computational resources available, and what effect the quality of the solution has on the overall system.

1.2.1.1 Heuristic Algorithms

Heuristic algorithms can be constructive or iterative [Breuer, 1972]. For example, consider the task of laying out a group of standard cells to minimize both the total length of the interconnection wires, and the net area occupied by the cells and wiring.

A *constructive algorithm* successively selects an unplaced cell, and adds this to a nucleus of already placed cells, until it has placed them all. Once it places a cell, the cell remains fixed. It chooses the next cell to be added to the nucleus by evaluating the length of wires needed to connect each cell to the nucleus, and evaluating the associated increase in area.

An *iterative algorithm* starts with an initial solution. It selects a set of cells to be moved along with a set of candidate locations, and evaluates the quality of the resulting solution. If the quality has improved, it accepts the solution. It continues this procedure until either the rate of improvement is low, or several iterations have occurred without improvement. Iterative algorithms have the potential of producing a better solution than constructive algorithms, at the expense of additional computer time. Both techniques may only find a locally optimum solution that could be far from the globally optimum solution.

1.2.1.2 Simulated Annealing

Simulated annealing is a technique that trys to avoid the local minima that conventional iterative methods produce [Kirkpatrick et al., 1983; Johnson et al., 1987; Rutenbar, 1989; Sechen, 1988]. It derives from the process used to strengthen materials. This process raises the temperature of a material so that the molecules of the material can move over greater distances. It then slowly cools the material down causing the molecules to lose their energy and restrict their movement. The resulting material has a more stable configuration. If the material cools quickly, irregularities in the material structure lock into place and result in a weaker material. This weaker material is analogous to the local minima obtained in the optimization problem.

Figure 1.4 shows the general form of the simulated annealing algorithm. The key feature of the algorithm is that there is a probability that it may accept a new solution with a higher cost. The current temperature and the magnitude of the cost difference determines what the probability will be. Hence at high temperatures it is possible to jump out of a local minimum. The algorithm slowly decreases the temperature until the chance of escaping from the current minimum is low. The resulting solution may still not be the optimum solution, but has a better chance of being closer to it than conventional iterative techniques. To get the best results, however, the algorithm needs to reduce the temperature slowly leading to long computation times.

1. Find an initial solution S (e.g using a constructive algorithm)
2. Calculate the cost C of S
3. Start with an initial temperature T
4. While T greater than final temperature do the following:
   (a) For a chosen number of iterations:
      i. Make a random, small change to the solution
      ii. Calculate the cost C^* of the new solution S^*
      iii. Calculate the cost difference $\Delta = C^* - C$
      iv. If $\Delta \leq 0$, $S = S^*$
      else set $S = S^*$ with a probability $e^{-\Delta/T}$
   (b) Decrease T

Figure 1.4: Simulated Annealing Algorithm

1.2.1.3 Knowledge-based Systems

Knowledge-based, or expert, systems help solve problems that have a complex cost function [Holden, 1987; Ackland, 1988; D&T, 1989a; Rosenstiel & Bergsträsser, 1986]. They try to emulate the methods used by an expert human designer. A collection of rules may represent these methods. They are not as efficient as mathematical approaches to the problem, and run slowly with large rule sets. Often, once an expert system is working, a software developer can gain additional understanding of the problem, and come up with a simple cost function that defines the mathematical relationship between the objects to be laid out. Hence, an efficient optimization method can use the cost function as a basis to lay out the objects. Expert systems also provide high level functions such as management of the overall design task, and evaluation of the overall manufacturability, testability and quality of the design.

1.2.2 Examples

The following is a brief overview of some of the design synthesis tools available. See Rubin [Rubin, 1987] for a more extensive discussion.

1.2.2.1 Cell generators

Cell generators produce a layout, from a set of specifications, based on a regular structure such as a Programmable Logic Array (PLA), a gate array, or a gate matrix.

PLA's consist of AND and OR planes of transistors. They generally occupy less area than purely random logic, but may be slower. Their main advantage is that software can generate them automatically. Techniques are available

to reduce the number of redundant gates to save area and to alter the size of transistors to meet speed specifications.

Gate arrays are more flexible than PLA's, and designers often use them for special purpose IC design. They consist of a fixed array of blocks, which contain transistor pairs, that the designer customizes to form logic gates such as AND, OR, and NOT. These blocks then interconnect to produce the circuit. A gate array generator produces the interconnection masks to customize the gate array design given a high level description of the circuit. The main disadvantage is that the interconnect routing can get complex, and long wire lengths can slow the circuit down.

Gate matrices have fixed vertical lines of polysilicon material that form the gates of transistors. The gate matrix generator produces the horizontal metal and diffusion wires to from transistors and their interconnections. The aim is to place transistors to reduce the length of the interconnection wires.

1.2.2.2 Compactors

Compactors minimize the area occupied by circuits designed using other design tools. They can achieve this, with minimal computational overhead, by squeezing the layout together along the x and y directions. They can also use optimization methods, as discussed in Section 1.2.1, to rearrange components to try to improve on the solution at the expense of extra time. Compactors free the designer, and higher level design tools such as cell generators, from the task of considering in detail the design rules of a particular fabrication process. Wolf and Dunlop [Wolf & Dunlop, 1988] discuss the compaction process in more detail.

1.2.2.3 Routers

Routers lay out the wires between cells, given a list of points to be connected. Simple techniques consider one wire at a time, and route it through a maze of obstacles. Other techniques consider multiple wires at a time and aim to reduce the area occupied by the wiring. Routers can use optimization techniques to make random changes to try to improve the solution. Lorenzetti and Baeder [Lorenzetti & Baeder, 1988] discuss routing in more detail.

1.2.2.4 Placement tools

Placement tools position cells so that they connect easily and occupy minimum area. Floorplanners allocate rectangular regions on the chip surface for cells that have yet to be fully designed. These may be expert systems that apply a series of rules, obtained from experienced designers, to decide how to arrange the cells. Expert systems can recognize well known structures in the design for which efficient structured floorplans are available. Standard cell placement tools place a set of standard cells, which may have a constant height and variable width, in rows with routing channels in between. They use optimization techniques such as simulated annealing to get a good solution [Sechen & Sangiovanni-Vincentelli, 1985]. Preas and Karger [Preas & Karger, 1988] discuss placement and floorplanning in more detail.

1.2.2.5 Silicon compilers

Silicon compilers convert a behavioural description of the circuit into a mask layout for fabrication. They start by converting the input description into a structural description such as a floorplan. Specialized cell generators produce each block of circuitry, and automatic routers generate the connections between them. Most of these compilers specialize in a particular type of

design, although they may have multiple floorplans to choose from. For example, they may specialize in signal processing using a specification of the algorithm as the input. They allow people without VLSI design expertise to produce IC designs. A truly general purpose silicon compiler has yet to be developed. Gajski and Lin [Gajski & Lin, 1988] discuss silicon compilation in more detail.

1.3 Static Analysis Tools

Static analysis tools analyse circuit properties that stay constant with changes in the electrical state of the circuit. They check fundamental electrical and geometrical rules of design, such as unintentional short circuits and minimum wire separations, as well as help the designer optimize the time critical sections of the circuit.

1.3.1 Algorithms

1.3.1.1 Layout analysis algorithms

Tools such as circuit extractors and geometric design rule checkers analyse the mask layout that serves as input to the fabrication process. The mask layout consists of a set of polygons for each mask layer. Its analysis falls into the field of *computational geometry* [Asano et al., 1986]. Algorithms to do this analysis must take into account the amount of main memory available to store the data structures, as well as their running time. Restrictions on the form of the layout, such as the use of *Manhattan* (or orthogonal) geometry, help to reduce the complexity of the task.

Pixelmap images (also called bitmap or raster images) can represent the mask layout. Each pixel has several bits: one for each mask layer. Algorithms to

analyse layout in this form are simple, but they require substantial storage space for a fine grid of pixels.

Tile-based (or corner-based) representations use the corners of regions, which consist of a combination of mask layers, to partition the plane into rectangular tiles. The layout analysis algorithms are similar to those that the pixelmap representation uses, but with lower storage needs. The storage requirements rise as the complexity of the layout rises. Both pixelmap and tile-based methods are only useful for Manhattan geometry.

Edge-based representations store the edges of the mask polygons, and sort them in order of their left end points and their slope. A *scanline*, which is a vertical line that scans across the plane from left to right, passes over these edges and analyses their relationships. Scanline algorithms only consider the edges that currently intersect with the scanline, thus reducing the number of edges that the main memory must contain in one instant.

Szymanski and Van Wyk [Szymanski & Van Wyk, 1988] give a good discussion of the above algorithms.

1.3.1.2 Network analysis algorithms

Tools such as network comparators and electrical design rule checkers analyse the electrical network of a circuit [Rubin, 1987]. Network comparison is an instance of the *graph isomorphism* problem. This problem has no efficient solution in the general case. *Graph partitioning* is the most common method of solving this problem. Some design systems restrict the designer, by placing limits on such things as the fan-in and fan-out of logic gates, to reduce the complexity of the network analysis. Szymanski and Van Wyk [Szymanski & Van Wyk, 1988] discuss network comparison algorithms in more detail.

1.3.1.3 Test generation algorithms

The number of possible combinations of circuit inputs grows exponentially with the number of inputs. It is impracticable to apply all possible input vectors to a circuit after manufacture to check that it has no faults. In general, the task of finding the minimal set of test vectors that will find all faults is NP-Complete [Breuer & Friedman, 1976]. Test generation algorithms try to find the smallest set of test vectors that will detect most faulty circuits.

There are several types of circuit faults [Gerner & Johansson, 1987]:

- *Static faults* change the function of a circuit and result from such faults as signals *stuck* high or low.
- *Dynamic faults* change the function of a circuit at certain frequencies or certain timing conditions.
- *Parametric faults* change the magnitude of circuit parameters such as voltages and currents.

Designers can choose test vectors by hand. This can be time consuming and prone to error. Another alternative is to use random sequences of test vectors, but their effectiveness decreases as the complexity of circuits increases [Bottorff, 1981].

The D-Algorithm [Roth, 1966; Roth, 1980] finds a test vector to detect a specified fault in a combinational logic gate circuit, by analysing the circuit description. It consists of three steps:

- *Fault sensitization*: find a set of input values that will produce a difference in the signal values between the faulty circuit and the good circuit at the fault location.

- *Fault propagation*: find a sensitized path through the circuit that propagates the difference to a circuit output.

- *Line justification*: find the values of the inputs required to meet the requirements of the first two steps by backtracing from the difference output to the circuit inputs. If a conflict in input values occurs, the algorithm must backtrack through the decisions it made in the first two steps.

The problem with the D-algorithm is that it can spend substantial time backtracking through decisions to resolve conflicts [Kirkland & Mercer, 1988]. Many circuits have many paths between the fault and a circuit output. *Redundant faults*, which are undetectable from outside the circuit, cause the D-algorithm to search all possible paths, which is time consuming. Jain and Agrawal [Jain & Agrawal, 1985] have modified the D-algorithm to handle transistor level circuit descriptions. Sequential circuits are hard to deal with as the algorithm needs to convert them into combinational form.

Some algorithms combine simulation with the procedure described above to reduce the time to generate the test vectors. For example, a fault simulator could find the other faults a particular test vector detects, hence reducing the number of times the D-algorithm needs to be run. Test generators use heuristics to try to reduce the amount of searching of the circuit structure needed to find a test vector. Kirkland and Mercer [Kirkland & Mercer, 1988], Fujiwara and Shimono [Fujiwara & Shimono, 1983], and Bottorff [Bottorff, 1981] give good discussions of some of these techniques.

Williams [Williams, 1981] states that the computational complexity of the task of finding test vectors, and finding their fault coverage, is $O(N^3)$, where N is the number of gates.

1.3.1.4 Hierarchical analysis

Static analysis tools can make use of the hierarchy in designs [Newell & Fitzpatrick, 1982; Scheffer & Soetarman, 1985; Stevens & McCabe, 1985]. By only analysing each cell once, tools can avoid needless and time consuming repetition. The technique relies on the designs being highly regular to offset the increased overhead in managing the hierarchy. Problems can occur when cells overlap, or wires pass through the cells, affecting the properties of a replicated cell in different positions in the design. Design systems often restrict the designer to using non-overlapping cells to improve the efficiency of the analysis tools.

Many of the static analysis tools flatten the hierarchy and treat the design as a single cell. This technique requires more space to store the complete design, but does not restrict the designer.

1.3.2 Examples

The following is a brief overview of some of the static analysis tools available. See Rubin [Rubin, 1987] for a more extensive discussion.

1.3.2.1 Circuit extractors

Circuit extractors derive from the mask layout an electrical network consisting of a list of transistors with their interconnections. This information is particularly useful when a designer has generated the mask layout manually from a higher level description, and wishes to confirm that it is correct. It is also useful for checking the output of an automatic synthesis tool. Circuit extractors can obtain detailed circuit information, such as parasitic capacitances and interconnect resistances, to use with analog circuit simulators.

For large circuits it can take a long time to extract the circuit.

1.3.2.2 Network comparators

Network comparators check that two networks from different sources are equivalent. For example, a designer can extract an electrical network from the mask layout of a circuit, and use a comparator to compare this with the desired network. They help the designer check a manually laid out circuit, and validate layout synthesis tools such as cell generators. They avoid the time consuming task of simulating the behaviour of the two networks, and checking that their results are equivalent.

1.3.2.3 Design rule checkers

Geometrical design rule checkers analyse the mask layout to check for problems that may lead to a low yield during manufacture. For example, wires that are too close together in the mask layout may become short circuits after fabrication. Each fabrication process has a set of rules for the mask geometry that allow for process errors; these include rules specifying minimum wire widths and minimum separation between features. Design rule checkers are essential for mask layouts that designers have produced manually. As processes become more complex, however, it has become increasingly hard for the designer to cope with these rules manually.

There has been a recent trend towards symbolic design, where designers use symbols for transistors and their interconnections, and use an automatic synthesis tool to generate the final mask layout. Although designers using automatic synthesis tools to generate the mask layout should not need design rule checkers, they still need them to validate the output of these tools. This is because the fabrication process technology often changes before the

synthesis tools are fully reliable.

1.3.2.4 Electrical rule checkers

Electrical rule checkers check for circuit errors that are detectable from an analysis of the electrical network of the circuit. Labels can identify each electrical node in the circuit. If an electrical node has two or more labels, this suggests a short circuit; alternatively, if a label refers to two or more nodes, this suggests an open circuit. If the labels include information on the electrical nature of the nodes, such as whether they are power rails, inputs, or outputs, more extensive checks are possible. Synthesis tools can generate much of this information automatically.

Electrical rule checkers can estimate the power consumption of the circuit by looking at the power requirements of each device, and estimating how many devices will be active at one time. The designer can use this information to check that the geometric size of the power rails is adequate, and to check that it meets the design constraints on power consumption.

1.3.2.5 Timing verifiers

Timing verifiers provide information on the *critical paths* of a circuit; these paths have the longest signal propagation times and determine the maximum speed at which a circuit can operate [Jouppi, 1987]. They do this by analysing the electrical network of a circuit and calculating how long a change in a signal will take to propagate through a block of circuitry. Timing analysis is independent of the state of the circuit and hence only needs to be run once for each design iteration. This contrasts with a simulator's approach where the state of the circuit determines whether a particular signal transition propagates through a block of circuitry. A simulation must run many times

with different inputs (or test vectors) and for most circuits it is impracticable to run all combinations of inputs to guarantee that it will find the critical paths.

1.3.2.6 Test generation tools

Test vector generators aid the designer in choosing a set of test vectors to apply to a circuit after manufacture. Automatic test equipment uses these test vectors to detect circuits that have manufacturing errors to minimize the number of faulty circuits delivered to customers.

Automatic test generators are available for combinational circuits. They produce test vectors to find static faults, especially stuck faults. Redundant faults that test generators find can aid the designer in simplifying the circuit. There are design techniques that incorporate circuitry to make it possible to test sequential circuits in a combinatorial mode [Gerner & Johansson, 1987; Williams. 1981]. Other tests, such as running the circuit at different clock rates, can find dynamic and parametric faults.

The time spent in finding a good set of test vectors can pay off in reducing the time needed to find faulty circuits, and reducing costs in repairing entire systems affected by a faulty circuit.

1.4 Dynamic Analysis Tools

Dynamic analysis tools (or *simulators*) determine a circuit's behaviour as a function of time after applying a set of inputs (*test vector*) to the circuit. They allow the designer to check the functional behaviour of a circuit, and to pinpoint timing problems, before fabricating the IC. They have become increasingly important as the complexity of IC designs has increased. Their

goal is to remove the expensive and time consuming IC fabrication step from the design cycle. Hon [Hon, 1987] and Terman [Terman, 1987] give good overviews of these tools.

1.4.1 Algorithms

There is a trade off between speed and accuracy in choosing an appropriate algorithm for dynamic analysis. A circuit consists of a set of nodes and branches; a branch forms when a circuit element connects two nodes. In high level representations of the circuit, the circuit elements correspond to functional blocks, whereas in low level representations, they correspond to devices such as transistors. A simulator uses models for the circuit elements to evaluate the node voltages at a given time. The accuracy of a simulator depends on:

- the accuracy of the circuit element models,
- the accuracy of the solution to the set of equations describing the voltage at each node,
- and the size of the time step between node evaluations.

By taking advantage of certain characteristics of a design technology, a simulator can improve its speed without losing accuracy. For example, many digital MOS circuits only have a small percentage of nodes with changing voltages at any time; this allows some sections of a circuit to be simulated with larger time steps reducing the total time to simulate the circuit.

1.4.1.1 Direct methods

The most accurate simulators work with the device level representation of a circuit, and produce analog waveforms for the voltages at specified nodes.

They use *direct methods*, as Figure 1.5 illustrates, to solve a set of circuit equations based on Kirchhoff's voltage law, Kirchhoff's current law, and the device characteristics [Sangiovanni-Vincentelli, 1981]. These are generally *non-linear differential equations* because of factors such as the non-linear characteristics of the *pn* junction in transistors, and the energy storage characteristics of capacitors.

Simulators cannot find a closed form solution to the circuit equations, so they use a *numerical integration* scheme that evaluates the node voltages at a series of time steps [Terman, 1987]. *Explicit* schemes, such as Forward Euler, rely only on values calculated at previous time steps, however they are numerically unstable. For accuracy, simulators usually use *implicit* schemes, such as Backward Euler. These schemes use an initial guess for the node values at the current time step, based on information from a previous time step, then improve this guess by iteration. This results in a set of *non-linear, difference equations*, which are *coupled* because each equation may depend on the solution of the other equations.

Direct methods solve the set of coupled equations directly using the *Newton-Raphson* method, which is a fixed point iteration scheme. The Newton-Raphson method converts the non-linear, difference equations [Chua & Lin, 1975] into a set of coupled, linear equations represented in matrix form. Forming this matrix involves evaluating the device models, and can be time consuming when using complex device models. The matrix is sparse, as for large circuits typically less than 2% of entries are non-zero. Simulators use sparse matrix versions of Gaussian Elimination or LU Decomposition, which try to avoid storing non-zero matrix elements and avoid performing operations with zero entries, to solve these equations in $O(N^a)$ time, where $1.1 < a < 1.5$ [Newton & Sangiovanni-Vincentelli, 1984]. They check the solution for convergence, and, if necessary, do another Newton-Raphson iteration. After the solution has converged, they select the next time step and

1. Apply nodal analysis to the circuit description and form the set of equations with node voltages as unknowns.

2. Begin with time $t = 0$.

3. While t is less than final time do the following:

   (a) Update the value of the circuit inputs.

   (b) Update the circuit parameters for the new time step.

   (c) Apply numerical integration to the *linear* differential equations.

   (d) While the solution has not converged, do the following:

      i. Apply numerical integration to the *non-linear* differential equations to form non-linear difference equations.

      ii. Apply the Newton-Raphson method to the non-linear difference equations.

      iii. Form the sparse matrix describing the set of linear equations.

      iv. Solve the linear equations by applying either Gaussian Elimination or LU Decomposition.

      v. Test the solution for convergence.

   (e) Check the local truncation error and save the solution if it is within bounds.

   (f) Choose the next time step.

Figure 1.5: Direct Method Algorithm

re-solve the equations.

1.4.1.2 Relaxation methods

The direct method for finding the node voltages requires extensive computation, hence it is only capable of handling designs with a few thousand transistors. Its main advantage is it copes with any circuit design. Relaxation methods attempt to reduce simulation time by decoupling the simultaneous circuit equations, solving each equation independently, and iterating the solutions until they converge. Decoupling the equations allows a simulator to evaluate some sections of the circuit with different time steps, and hence take advantage of circuit latencies. Simulators can apply these techniques at various stages of the solution of the circuit equations. Newton and Sangiovanni-Vincentelli [Newton & Sangiovanni-Vincentelli, 1984; Newton, 1981] give a good overview of these methods.

The two main relaxation algorithms are *Gauss-Jacobi* and *Gauss-Seidel.* Gauss-Seidel uses node voltages that it has already calculated in the current time step, and thus converges faster than Gauss-Jacobi, which only uses values from the previous time step in solving each equation. The order in which the Gauss-Seidel method evaluates the node voltages affects how fast the solution converges. Gauss-Jacobi is suitable for executing in parallel, because at each time step it solves the equations independently in any order. These methods can decouple the set of linear equations, resulting from the application of the Newton-Raphson method, and solve them in $O(N)$ time, instead of directly solving the sparse matrix. Usually, however, they decouple the equations at the non-linear equation level; this eliminates the time consuming step of applying the Newton-Raphson method to a set of coupled equations. They then use the Newton-Raphson method to solve each of these equations without the need for matrix arithmetic.

Some simulators get substantial speed improvements by only applying one relaxation iteration, and using a small time step to reduce errors. They are effective for simple circuits, such as in gate array designs, which are without feedback loops, pass transistors and floating capacitors [Newton & Sangiovanni-Vincentelli, 1984].

Iterated timing analysis uses one Newton-Raphson iteration for each non-linear equation, and carries the relaxation method to convergence [Newton & Sangiovanni-Vincentelli, 1984]. This gives accurate results, but for tightly coupled circuits may take longer than direct methods. The convergence criteria determines the speed and accuracy properties of these methods.

Waveform relaxation applies relaxation techniques to the differential equations by decoupling the equations, and solving each equation over the entire interval of the simulation [Lelarasmee et al., 1982].

1.4.1.3 Event driven simulators

Simulators can use a time step that the designer sets, or they can set the time step dynamically depending on the convergence properties of the solution. *Event driven simulators* maintain a time ordered queue of events. They remove from the queue all events that occur at the same time as the event at the head of the queue, and evaluate a new node voltage for each node that an event refers to. If the voltage of a node changes, they place a new event on the queue for each node that it connects to, with a time equal to the current time plus the node's response time to the current event. This process continues until there are no more events on the queue or it reaches the end of the simulation interval. Event driven simulators do not spend time evaluating nodes that are stable, however, they may spend excessive time evaluating events that are unimportant. They suit large digital circuits where only a small percentage of the nodes are changing at any point in time.

Hon [Hon, 1987] discusses them in more detail.

1.4.1.4 Device models

The choice of device model affects the speed and accuracy of a simulator [Terman, 1987].

Analytic models are highly accurate and need detailed information on such things as physical device geometries, electrical properties of the materials used in the fabrication process, and temperature of operation. They are suitable for looking at the effects of fabrication parameters on the circuit performance, and can produce data for approximate models. Simplified models assume many parameters are constant; for example, they may treat the voltage dependent gate capacitance of transistors as constant, but they still require the evaluation of complex functions.

Empirical models try to find simpler functions to speed the evaluation of the device model by using curve fitting techniques.

Table driven models do not require the evaluation of functions, which improves the speed substantially, but they require more memory to store the tables. Simulators use interpolation between the table entries to achieve improved accuracy.

1.4.2 Examples

The following is a brief overview of some of the types of simulators available. See Hon [Hon, 1987] for a more extensive discussion.

1.4.2.1 Circuit simulators

Circuit simulators produce detailed analog waveforms of the node voltages of a circuit in response to a set of inputs. The designer chooses which nodes are of interest, and how much real time the simulator shall cover. A circuit extractor can supply the simulator with the necessary circuit parameters, such as a list of transistors with their connections. the size of the transistors, the gate capacitances, and the parasitic capacitances. Simulators that use direct methods are the most accurate and reliable, but their long running times make them only suitable for small critical sections of a large circuit.

Other simulators use relaxation techniques, such as iterated timing analysis and waveform relaxation. to reduce the running time for typical digital MOS circuits [Newton. 1979; Newton & Sangiovanni-Vincentelli, 1984]. For simple circuits, they run at least ten times faster than simulators that use direct methods. They break the circuit up into decoupled subcircuits, which they can simulate with different time steps depending on activity, and use event driven simulation to take advantage of circuit latencies.

1.4.2.2 Timing simulators

Timing simulators produce the same information as circuit simulators, but at reduced accuracy [Chawla et al., 1975; Newton, 1981; Ackland & Clark, 1989]. They can run more than a hundred times faster than direct method simulators. They use relaxation methods, usually without iterating the solutions to convergence, with approximate device models that are often table-driven. Designers have successfully used them on digital MOS circuits, with limitations such as requiring all capacitors in the circuit description to have one terminal connected to a reference, and restricting the use of pass transistors and feedback loops. They are useful for checking the operation of the

complete circuit, and pin pointing areas that may need a circuit simulator to check in more detail.

1.4.2.3 Logic simulators

Logic simulators evaluate the logic states of each node of a digital circuit in response to a set of inputs. The logic states usually include high (1), low (0), undefined (X), and high impedance (Z). Murai [Murai et al., 1986], and Newton [Newton, 1981], discuss the types of logic simulators and the algorithms that they use.

Switch level logic simulators treat the circuit as a collection of transistors and wires [Bryant, 1984]. They model transistors as simple switches with a simple delay model, and they model wires as zero delay conductors. They provide simple timing information to help designers find possible timing problems in a circuit. Transistor level descriptions are available from either circuit synthesis tools or circuit extractors.

Gate level logic simulators treat the circuit as a collection of interconnected logic gates. They are simpler and hence faster than switch level simulators, but cannot easily handle pass transistors and tri-state bus structures. They model the behaviour of gates using truth tables. Their main use is in verifying that the circuit is functionally correct, making them useful for gate array design. For large circuits, designers have usually created a schematic description that they can use as input to a logic simulator. Logic simulators are usually event driven to reduce the simulation time. Internally they can use extra states to model the effects of pass transistors and wired gate logic. Some simulators distinguish between states (such as high and low) and strengths (such as driving, resistive and high impedance).

1.4.2.4 Fault simulation

Fault simulation checks how many manufacturing defects, such as circuit nodes shorted to ground, a given set of test vectors would pick up during the testing phase of IC manufacture. By deliberately introducing defects into a logic-level circuit description, a fault simulator can run a simulation with the set of test vectors to see if the outputs of the circuit show any indication of a fault. The *fault coverage* of a set of test vectors is the ratio of errors discovered to the number of possible errors. A designer can use this information to improve the testability of a circuit before detailed design, and to choose a suitable set of test vectors for use by the manufacturing process.

Fault simulators use similar techniques to event-driven logic simulators, with a few extensions to simulate many faults during a single simulation run. Bottorff [Bottorff, 1981] describes three different approaches to fault simulation: parallel, deductive and concurrent.

Parallel fault simulation uses bits in a computer word to represent the signal values of the fault free and several faulty circuits. Word operations allow a computer to simulate several circuits with different faults simultaneously; large numbers of faults require multiple simulation runs. Parallel fault simulation has a time complexity of $O(N^3)$, where N is the number of logic gates [Bottorff, 1981].

Deductive fault simulation [Armstrong, 1972] only simulates the behaviour of the good circuit, and deduces from the current state all the faults detectable at any internal terminal or circuit output terminal. The algorithm calculates fault lists at the output of each logic element. Faults in this list arise from faults associated with the element's inputs, that for the current logic state cause an erroneous element output. The algorithm also adds faults, occurring within the element, to the resultant fault list. When unknown or undefined states (X state) are present, however, the calculation of the fault lists is more

difficult. Deductive fault simulation has a time complexity of $O(N^2)$, where N is the number of logic gates. It uses more memory than parallel fault simulation, and may need several runs with truncated fault lists.

Concurrent fault simulation combines features of parallel and deductive simulation [Bóttorff, 1981]. The algorithm simulates faulty circuits only when a fault causes a gate input or output to differ from the good machine. This contrasts with the parallel method, which simulates both good and faulty circuits even when the logic values are equal, and contrasts with the deductive method, which only simulates the good circuit and deduces the fault machine behaviour from the current good machine state. Concurrent fault simulation uses fault lists for each gate, and schedules events when changes in logic values occur in the good machine, or in the fault machines in the fault lists. It uses more storage than the deductive technique, but can handle mixed level representations and nominal delay simulation [Gai et al., 1988]. It also has a time complexity of $O(N^2)$.

Some test generators and fault simulators can use functional-level circuit descriptions [Bottorff, 1981]. They require that the functional primitives have a high structural regularity, such as RAM arrays.

1.4.2.5 Functional simulators

Functional simulators analyse high level descriptions of a circuit, where the circuit elements consist of functional blocks in the design hierarchy. These functional blocks have a mapping to the hardware construction of the circuit. These simulators provide similar information to a logic simulator; they check that the circuit will function correctly for a range of input data. They are fast and allow the designer to quickly simulate the complete circuit at early stages in the design process. The input to a simulator is a high level programming language description of the circuit. Structured programming

techniques can represent the functional hierarchy by using subroutines to describe the function of each block, and using parameter passing to specify the block interconnections. The functional descriptions are simple, and are analogous to the truth tables of gate level simulators.

1.4.2.6 Behavioural simulators

Behavioural simulators analyse detailed high level descriptions of a circuit's behaviour. The structure of the behavioural description may be without a direct mapping to the hardware construction of the circuit. Like functional simulators, the circuit description can be in a high level language, but it contains more detailed information including timing information. Manufacturers sometimes supply behavioural descriptions of their circuits to allow designers to integrate them into their own systems.

1.4.2.7 Mixed level simulators

Mixed level simulators accept circuit descriptions with different levels of representations [Rammig, 1986; De Man et al., 1981]. This allows a designer to examine in detail the response of particular sections of a circuit to a set of external input signals, without incurring the overhead of simulating the whole circuit at that detail. A designer may combine a recently designed transistor level description of a circuit sub-block, with a functional description for the rest of the circuit. A mixed level simulator can simulate this sub-block at the switch-level in the context of the whole circuit.

1.4.2.8 Mixed mode simulators

Mixed mode simulators simulate a circuit description at different levels of detail [Schindler, 1987; Newton, 1981; D&T, 1989b; Overhauser et al., 1989].

For example given a transistor level description of a circuit as input, a mixed mode simulator could simulate some sections of it at the analog level, and other sections at the digital, or logic, level. They have the same advantages as mixed level simulators: often simulators support both features and use the two terms interchangeably. Mixed mode simulators are useful for circuits that combine analog functions, such as amplifiers, with digital functions, such as registers.

1.5 Summary

There are many computer aided design tools available to aid the designer in designing a circuit. They achieve the computer aided design objectives of minimizing design errors, and aiding design changes. As design tools automate more of the tasks traditionally done by the designer, the time taken to produce the design will become dominated by the running time of the design tools.

Ideally a circuit will meet the design specifications the first time the designer gets it fabricated, but typically the designer requires two passes of the fabrication step before the circuit is available for use in its intended application. For large volume production, however. it is economical to further refine the design to increase the yield, and hence reduce the production costs.

To avoid multiple passes of the fabrication step, a circuit requires thorough design verification. Although static analysis tools such as timing verifiers can do some of this, verification requires extensive simulation of the circuit at all stages of design.

Simulation is the most time consuming step in the design cycle. Each simulation run must evaluate the state of the circuit at many time points, and many simulation runs with different input data are necessary for complete

verification. Each time a design changes, the designer must resimulate the circuit to ensure the change was successful. The time required to obtain the results of a simulation should be short enough so that the details of the design are still fresh in the mind of the designer. Present uniprocessor workstations only accurately simulate small sections of a large circuit in a sufficiently short time.

To meet the computer aided design objectives of shortening the time for design synthesis and verification, design tools must run on computers faster than present general purpose workstations. Faster computers will reduce the time a designer needs to design a circuit that meets its specifications, and thus make it more economical to use complex integrated circuits in low production volume applications.

Attempts to increase the speed of computer systems for circuit design have ranged from using special purpose hardware, to using distributed workstations working together in parallel. General purpose parallel processing systems have become cost effective because of advances in microprocessors and interconnection techniques. These systems have the advantage of flexibility, and offer substantial reductions in running time for design tools that exploit the parallelism available in their algorithms.

The design tools available for computer aided design cover a wide range of computational techniques. Although special purpose hardware may aid a few of these techniques, the design system must be a flexible environment that can efficiently run a wide range of design tools.

The *goal of this thesis* is to propose a general purpose parallel computing system that will support the full range of design tools, and offer a substantial speed up to the design verification step in the design cycle.

Chapter 2 will describe work done by the author on building a processing system for the real time acquisition and control of external signals, and the

implementation of a parallel circuit extraction algorithm on a multicomputer. This work generated initial ideas for a multiple processor computer system for running CAD software.

Chapter 3 will begin by overviewing the various approaches to designing fast computers, and indicate which approaches are best for accelerating CAD software. It concludes that a Multiple Instruction stream / Multiple Data stream (MIMD) architecture offers the highest flexibility. It goes on to analyse recent research into parallel CAD algorithms on MIMD architectures, before recommending a heterogeneous multicomputer architecture for running CAD algorithms.

Chapter 4 discusses hardware and software requirements of the multicomputer proposed for running CAD algorithms. It includes sections on the computing node architecture, the interconnection network, the secondary storage system, the operating system, and the programming environment.

Chapter 5 presents the conclusions reached in the thesis.

Chapter 2

Background

The ideas for a general purpose parallel computing environment for computer aided design to be presented in Chapter 4, derive from the author's experiences with both hardware design for high performance processing, and parallel processing software development for computer aided design.

The hardware design project, carried out early in the thesis work, consisted of the design and construction of a real time subsystem (RTS) for a VAX™ 11/780 super-minicomputer used for University research. This subsystem provides researchers with low cost real-time acquisition and control of external signals, which allows computer control of laboratory experiments and direct processing of results. The design provided experience with a specialized distributed processing system, and with high speed communications.

The hardware development led to an interest in parallel computer architectures for providing cost effective performance beyond that achievable by single low cost microprocessors. The University of Adelaide's strong background in teaching VLSI design and developing computer-aided design software provided incentive to looking at using parallel processing to speed up the time consuming stages of computer-aided design.

This led to an investigation into the feasibility of using a general purpose, message based multiprocessor to speed up the time consuming circuit extraction phase of circuit simulation. This investigation provided experience with programming in a practical parallel processing environment, and highlighted some of the limitations of parallel processing architectures.

This chapter details the work on the RTS and on the parallel circuit extraction software, and points out the ideas for a general purpose parallel computing system for computer-aided design that arose from the work.

2.1 Real Time Subsystem (RTS)

2.1.1 Introduction

The Department of Electrical and Electronic Engineering at the University of Adelaide, Australia required a general purpose computer system capable of acquiring, and responding to, large bandwidth external signals. Researchers wanted a system for conducting experiments that could

- acquire data from external systems at high sampling rates,
- provide real-time pre-processing of external data before storing it for later off-line analysis,
- output external control signals that can respond to external events in real time,
- and provide a user friendly computing environment for later analysis of results.

At the time, the Department had a DEC VAX[1] 11/780 running the UNIX[2] 4.3 BSD operating system [Quarterman et al., 1985]. This system included

- secondary storage for storing results for future analysis,
- mathematical software for doing matrix arithmetic,
- graphics packages for displaying results on monochrome graphics terminals and producing hardcopy output on plotters,
- and software development support for writing programs to analyse results.

Researchers used this system for general text processing, software development, and simulating complex systems.

The VAX™ has a bus called UNIBUS™ for interfacing with external peripherals such as disk drives, printers, and terminals. Researchers could have added modules to this bus to allow the input and output of external signals for use in experiments, but the system was unsuitable for responding to external events that occur at high frequency (greater than 100kHz). Real-time processes would have had to compete with other users for CPU time and I/O access. If these real-time processes had higher CPU priority over other processes, they would significantly degrade the response time for interactive users, and the real-time response would still be unpredictable.

The next section discusses the characteristics of real-time systems and discusses the overheads in a general purpose system that prevent it from being suitable for responding to critical external events.

[1]DEC, UNIBUS and VAX are trademarks of Digital Equipment Corporation

[2]UNIX is a trademark of AT&T Bell Laboratories

2.1.2 Real Time versus General Purpose Computer Systems

2.1.2.1 Real time computer systems

The correct operation of a real-time computer system depends not only on the calculation of a correct response to an external event, but also on the time it takes to respond to the event [Stankovic, 1988]. For example, in a real-time control system a computer might generate a control signal, and monitor some parameter of an external system. After comparing this parameter with a reference value, the computer may decide to change the value of the control signal to correct the parameter before damage occurs in the external system. If the computer does not calculate the new value for the control signal quickly enough, the system will fail.

It is important that a computer in a real-time environment can guarantee to meet the timing requirements of an external system; hence the computer's timing behaviour needs to be predictable.

Stankovic [Stankovic, 1988] gives a good overview of the issues involved in real-time computing.

2.1.2.2 General purpose computer systems

General purpose computer systems, such as the VAX™ 11/780 running the UNIX™ operating system, try to minimize the average response time for many tasks.

Operating systems achieve a high CPU utilization rate by switching to another task when the current task is waiting for I/O. This may happen when a task is waiting for data from secondary storage, or a response from a user's terminal.

To make effective use of limited primary memory, operating systems *swap* processes, which have been idle, from primary memory to secondary storage, and retrieve them when the processes are ready for execution again. Operating systems also use *paging* techniques to limit the primary memory required by a process to the set of pages needed for the current series of operations. As new pages are read in from secondary storage, the least recently accessed pages are written back.

Real-time processes may remain idle for long periods of time before receiving an interrupt. When they do receive an interrupt, they may need to service the interrupt quickly. The operating system, however, may need to swap the process back in or retrieve pages from secondary storage before servicing the interrupt. This leads to unpredictable response times.

In a *multiuser* system, the operating system tries to distribute the CPU time fairly among a set of tasks. It does this by allocating time slices to each process: when a process's time slice is up, the operating system schedules another process for execution. This prevents a user from stopping other users getting responses from interactive programs, such as editors, by running a CPU intensive program that does not need frequent access to an external device. The priority of a task determines the size of a time slice, and the frequency at which the operating system assigns it to a task. The operating system can dynamically alter this priority during the execution of a task, leading to unpredictable response times.

Some multiuser operating systems include features to make the response time to external events more predictable. For example, Masscomp's modified version of the UNIX™ operating system allows real-time processes to have fixed CPU priorities [Blackett, 1983]. The highest priority real-time task gets unlimited CPU time until it completes, pauses, waits for I/O, or a higher priority process interrupts it. When the current real-time process

waits for I/O, the operating system will schedule the real-time process with the next highest priority. While there are no real-time processes waiting for execution, the operating system will distribute CPU time fairly among time-sharing processes in the normal way. To prevent memory management uncertainties, a real-time process can prevent the operating system swapping it to disk, or paging parts of it to disk. Real-time processes can also assign contiguous areas of storage on disks, for storing and retrieving information at a predictable, sustained data rate.

The following paragraphs discuss the main overheads incurred when switching between processes [Blackett, 1983] in response to an interrupt.

Interrupt latency within the operating system is the time between an external system asserting an interrupt, and the operating system accepting the interrupt. This latency results from the operating system switching off an interrupt while it executes a critical section of code.

Context-switch time is the time the operating system takes to stop one process and start another. During this time, the operating system saves enough information about the state of the current process to allow it to restart where it left off, and restores any previous state for the next process. The state of a process may include the contents of the CPU's registers, memory management information, and the status of open files. If the operating system needs to retrieve a process, or parts of a process, from secondary storage during a context switch, the context switch time increases significantly.

Although there are techniques for providing predictable response times for real-time events in a general purpose operating system, the overheads of interrupt latency and context switch time limit the rate at which the operating system can respond to interrupts. This rate is typically less than 10,000 interrupts/sec, which is inadequate for responding to external events that occur at rates above 100kHz. General purpose file systems also limit the rate at

which the system can store information data on disk.

2.1.3 Real Time System Overview

The goal of the real time subsystem (RTS) was to preserve the existing computing environment, provided by the VAX™ 11/780 general purpose computing system (GPS), and provide real-time facilities required for laboratory research. Figure 2.1 shows a block diagram for this system.

The RTS consists of a single board computer (SBC) that communicates with external interfacing modules over a standard backplane. These modules include a 16-channel Analog/Digital (A/D) converter module, an 8-channel Digital/Analog (D/A) converter module, a Timer/Parallel digital Interface (TPI) module, and a datalink to the GPS. Figure 2.2 shows a block diagram of the architecture of the RTS.

The SBC contains no operating system. The GPS provides the facilities for program development, and downloads programs to the SBC. A series of software routines are available on the GPS to assist programmers to interface with the hardware available on the RTS. The simple software system removes the overheads that are present with general purpose operating systems, such as file system management, memory management, and process scheduling. To respond to an external interrupt, the RTS only needs to save the CPU registers and program counter, thus minimizing the response time to an interrupt. Only one user at a time will use the RTS, so only programs needed for the current experiment reside in the RTS. Programs remain in main memory for the duration of an experiment, thus removing paging and swapping overheads.

The RTS can acquire results and send them to the GPS for later analysis using the mathematical software available on the GPS.

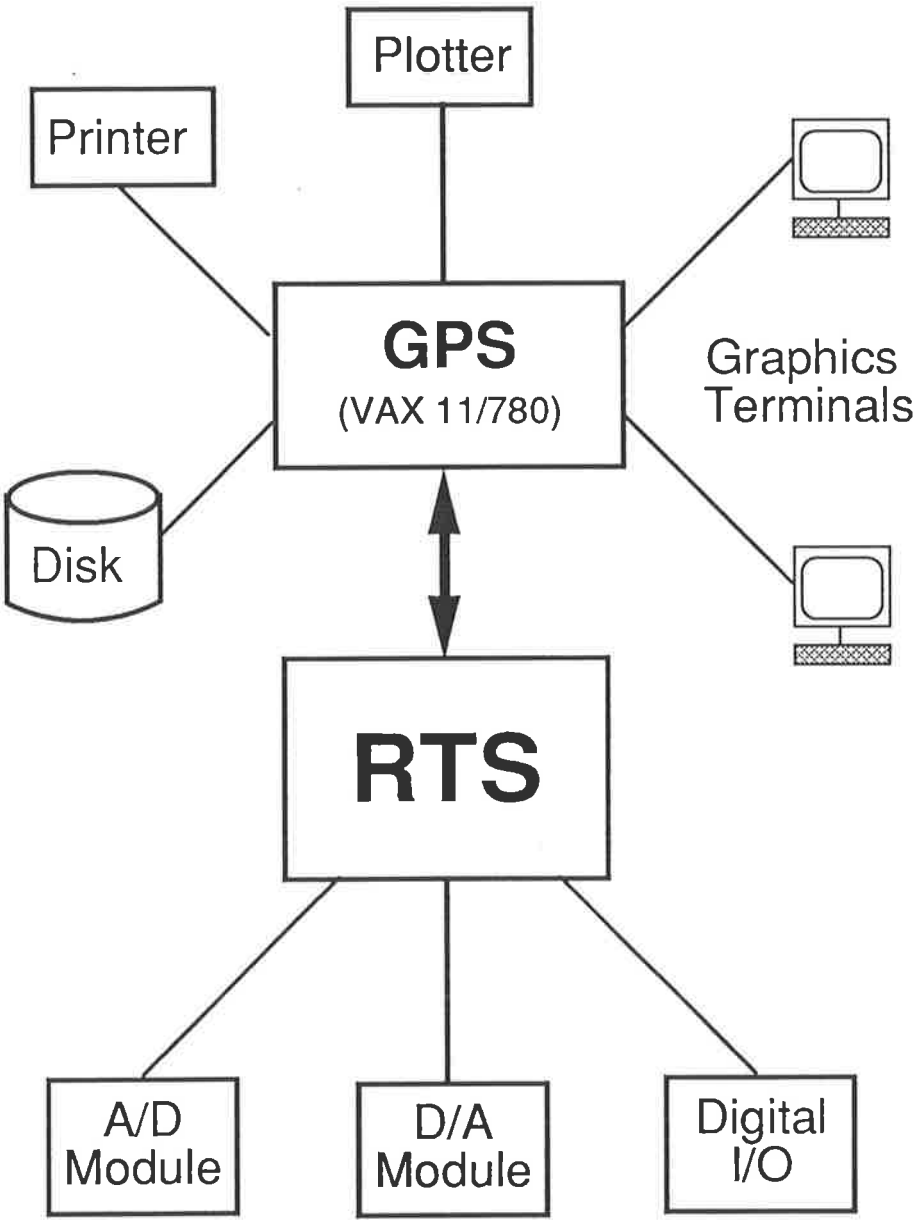


Figure 2.1: System Overview

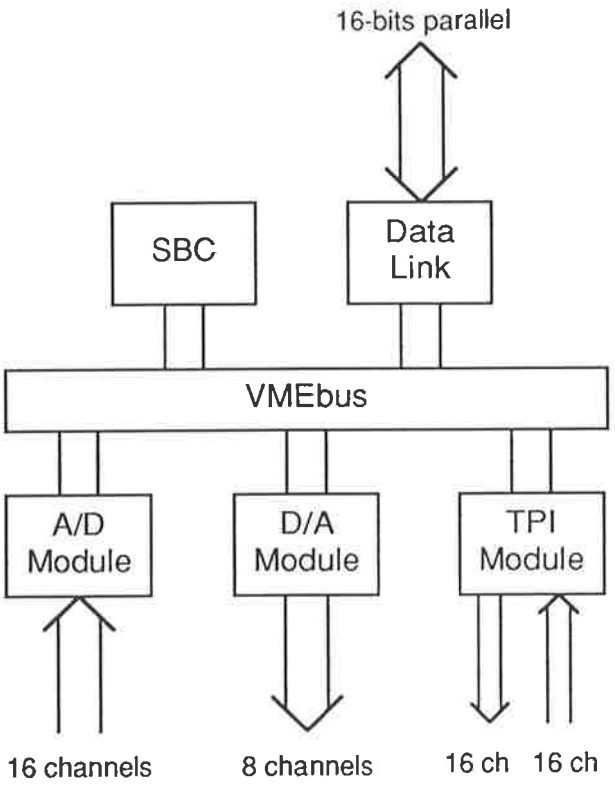


Figure 2.2: Real Time Subsystem (RTS) Architecture

Appendix A describes the system in more detail.

The next section will discuss ideas for the hardware and software characteristics of a multiple processor computer system suitable for running CAD software, which arose from work developing the RTS.

2.1.4 Conclusions

The development of the RTS has provided some initial ideas for, and insights into, the structure of a high performance computing system for computer aided design. The basic idea is that the system should consist of both a *general purpose computing section* (GPCS), and a *special purpose computing section* (SPCS). Designers will use the GPCS for the user interactive stages of design, such as circuit layout, and also for the general management of the design using facilities such as text editors. The SPCS will allow designers to off-load the compute intensive design tasks, such as circuit simulation, from the GPCS, to reduce the running time of these tasks. The aim is to provide a high performance system that is affordable for small design teams. This is in contrast to many large scale supercomputers that provide high performance that only large corporations can afford.

The following is a list of recommendations for the high performance computer system derived from the development of the RTS:

- use high performance, high production volume, commercial microprocessors,
- use as much local memory as possible,
- use open standards where possible to take advantage of other manufacturers expertise and to reduce costs.
- place the main user interaction facilities on the GPSC,

- use a widespread operating system for the GPSC,
- support well known and understood programming languages,
- keep the operating system software on the special purpose section as simple as possible.
- allow the user software full access to the low level facilities available on the special purpose section as well as offer simple high level routines,
- and provide high speed communications between the GPCS and the SPCS to allow the exchange of large programs and data.

The use of commercially designed microprocessors substantially reduces the development costs of a high performance design. The best performance/price ratio usually occurs when using a microprocessor that is in high volume production. This implies that the microprocessor will be general purpose, and not necessarily optimum for a particular application. It will have the advantage of a large base of development tools from different manufacturers, and have readily available compilers for a range of languages. There are often support integrated circuits (ICs) available for high production microprocessors, including ICs for memory management, and floating point acceleration. To achieve the same performance as a highly optimized single processor supercomputer, a system will require multiple microprocessors working together in parallel.

A system with a large amount of local memory minimizes references to non local storage, such as disk storage, and reduces the need for complex memory management. To put the maximum amount of memory on the same board as the CPU, hence reducing access delays, a system needs to use the highest density memory available. The access speed of high density memory is too slow for recent microprocessors, which can operate with clock rates above 40MHz, but the use of cache memory can overcome this mismatch.

The use of commonly used open standards in the design takes advantage of the large base of products available that are compatible. Parts of a design that can use these standards include the bus structure used to connect the different elements of the design, and the interfaces to standard devices such as disk drives. The competition between manufacturers producing equivalent items, and the larger volumes of production, reduce the price/performance ratio for essential items such as memory boards, CPU boards, disk controllers and disk drives. Incorporating these items into the design saves on design time, and allows designers to take advantage of specialist expertise from other manufacturers. The disadvantage is that these products are not optimum for a particular design, and may contain functions that are unnecessary. By customizing a design, higher performance is possible at the expense of the extra design and development time.

By placing the main user interaction facilities on the GPCS, the system can optimize the design of the software system on the SPCS for executing the compute intensive stages of design. This allows a user to benefit from a sophisticated user interface without compromising system performance.

By using a widespread operating system, such as the UNIX™ operating system [Ritchie & Thompson, 1974; Quarterman et al., 1985; Blair et al., 1985], as the basis of the operating system for the GPCS, the high performance computing system can take advantage of an environment that many users are familiar with, and for which there is a large software base. Features that such an environment provides include command interpreters, text editors, text formatters, file management, and software development facilities. Users are more likely to use a system that has a human/computer interface that they are familiar with. The underlying details of the operating system can be different from the generic version, but the programmer's interface should appear the same as for a standard operating system.

By supporting well known languages, such as C and FORTRAN, for programming both the GPCS and the SPCS, the system can encourage programmers, who are already expert in these languages, to produce software for the system. It also allows programmers to easily port popular software, already written in these languages, to the new machine. Designers can add extensions to these languages to harness the special abilities of the high performance system. The compilers for these languages produce reliable output, and incorporate advanced optimization techniques.

By keeping the operating system software on the SPCS as simple as possible, the system allows software developers to develop their own optimized versions of facilities such as file management and memory management for running specific applications. The operating system does not need to have facilities for general purpose computing, such as fair process scheduling with automatic swapping out of idle processes, that can add to the running time of applications.

By allowing software developers full access to the low level facilities available on the SPCS, the system lets them get the full benefit of the processing power available. A library of software routines can simplify access to these low level resources. High level software routines that behave at the level of general purpose operating system calls will allow software developers to produce prototype software quickly.

In addition to providing the software development facilities for the SPCS, the GPCS will provide for downloading programs with their input data, and receiving their results. This interaction will require high speed communications to prevent the data transfer time from overcoming the speed advantages of the SPCS.

2.2 Parallel Circuit Extraction

2.2.1 Introduction

Present uniprocessor general purpose computers, such as uniprocessor workstations, are inadequate for the compute intensive stages of computer aided design of VLSI circuits, such as circuit simulation. Although large scale single processor supercomputers can provide large speed ups over general purpose computers for vector based calculations, their cost is too high for most designers. For calculations that do not involve vector arithmetic, these supercomputers are not cost effective.

The alternatives available to designers range from using special purpose accelerators to using networked workstations working together in parallel. Special purpose accelerators usually only accelerate a few of the most time consuming stages of design, such as logic simulation [Blank, 1984; SIGDA, 1988; Lewis, 1988]. They offer the best performance but are inflexible; a new algorithm can make them redundant if it provides equivalent performance on a general purpose computer.

Networked workstations allow designers to use the excess computing capacity of other workstations. Typically each designer may have sole use of a workstation to provide both a fast interactive response and good graphics for the interactive stages of design, such as circuit layout. Often these workstations have idle computing capacity while waiting for designers to input their next instruction. By using this idle capacity in parallel, the compute intensive stages of design can run in the background at a faster rate than if a single workstation ran them exclusively. For this to be effective, it must be possible to partition the task in a way that minimizes the amount of communication over the network. Since this is not possible for all applications, distributed workstations do not offer a general performance improvement for compute

intensive tasks. Also running compute intensive applications in the background can degrade the response time for the interactive user, such as when pages of the interactive task are pushed out of primary memory. There are also problems of load balancing among workstations that have a dynamically varying load. Widdowson and Ferguson [Widdowson & Ferguson, 1988] discuss an application of networked workstations. Agrawal and Jagadish discuss methods for optimally partitioning a problem for running on a network of workstations [Agrawal & Jagadish, 1988].

The development of the RTS provided ideas for the structure of a high performance computing system for computer-aided design. This system relies on using high production volume commercial microprocessors, coupled together with a fast interconnection network, to achieve cost effective high performance. It has the advantage of flexibility, and the potential for providing large speed up factors when software developers exploit the parallelism of an application. Gordon Bell [Bell, 1989] presents a good overview of the cost effectiveness of multiprocessors versus traditional supercomputers.

Ackland [Ackland et al., 1985; Ackland et al., 1986] has shown that message-based general purpose multiprocessors can effectively speed up the simulation of a circuit. These multiprocessors adapt well to the task of circuit simulation because circuits operate by passing signals between subcircuits; by assigning these subcircuits to different processors, the evaluation of circuit operation can proceed in parallel.

Circuit extractors derive accurate circuit parameters for input into circuit simulators by analysing the circuit mask layout, which serves as input to the fabrication stage of design. The circuit extraction for a large circuit (greater than 100,000 transistors) takes several hours on a conventional workstation (such as a SUN 3/260). A designer must rerun the extractor after each circuit modification. Thus if circuit extraction is amenable to parallel processing,

a computer system could significantly reduce the time for designing a large circuit.

This section describes the use of a general purpose message based multiprocessor (also called *multicomputer* as multiprocessor usually refers to a shared memory multiple processor architecture) to speed up the task of VLSI circuit extraction [Tonkin, 1990]. This work began in February 1988 at AT&T Bell Laboratories, Holmdel, NJ, USA. At the time, researchers had not done much work in this area since circuit extraction is not as amenable to parallel processing as other design tasks such as circuit simulation. The aim was to show that a multicomputer could offer significant speedups (*i.e.* > 10) for tasks other than circuit simulation. This will help show the potential of using parallel processing for speeding up the compute intensive integrated circuit design tasks, and will provide additional information on the facilities needed on a multiple processor machine to support these tasks.

2.2.2 Overview of Circuit Extraction

Circuit extractors derive from the circuit mask layout a list of transistors along with net numbers for each transistor's terminals, and the length and width of each transistor's channel. Connected transistors will have the same net number on corresponding terminals. Extractors match the labels assigned to electrical nodes by the designer with the extracted electrical nodes. Logic simulators can use this information to verify the logical operation of a circuit.

The electrical path between transistor terminals comprises several regions that consist of different combinations of mask layers. These regions have differing electrical properties. Extractors calculate the area of each region, along with the region's perimeter with each region touching its boundary. A simulator can use this information to determine the capacitance associated with each electrical node in the circuit.

The circuit layout usually consists of a hierarchy of cells. Cells consist of other smaller cells, and collections of polygons representing different mask layers. Researchers have taken advantage of this hierarchy to speed up layout analysis [Gregoretti & Segall, 1984], by analysing each cell only once. Problems occur when the cells overlap, hence altering the internal behaviour of the cells. Solutions to this either involve restrictions on the designer such as requiring the use of non-overlapping cells [Stevens & McCabe, 1985; Scheffer & Soetarman, 1985], or involve techniques that may result in a longer running time for irregular designs [Newell & Fitzpatrick, 1982].

The alternative is to flatten out the hierarchy and work on the mask layout as a single cell. It has the disadvantage of requiring a large amount of memory to store the input mask description in its flattened form. For example, a 130,000 transistor design required 6 megabytes to store the hierarchical description, and 56 megabytes to store the flattened description. This work uses the flattened description of a circuit, as it applies to all design techniques, and shows how a multicomputer can handle such large quantities of data through the use of secondary storage.

2.2.3 Goalie

Goalie is a circuit extractor developed by Szymanski and Van Wyk [Szymanski & Van Wyk, 1985] to extract circuit information from large mask layouts. It uses edges to represent the mask polygons. Each edge has information on the coordinates of its left and right endpoints, its electrical net number, its mask layer, and the side of the polygon it represents. Goalie supports non-orthogonal geometry.

The *first step* in the extraction flattens the hierarchical circuit description into a set of edge files, one for each mask level. It sorts the edges in order of their left endpoints and on their slope.

The *second step* creates a series of intermediate layers that represent the resultant electrical characteristics produced by the boolean combinations of the original mask layers. Examples of the intermediate layers (for a MOS circuit) are transistor layers, polysilicon wire layers, diffusion wire layers, metal to polysilicon contact cut layers, and metal to diffusion contact cut layers. It can assign unique numbers to the edges of each separate transistor region.

The *third step* assigns unique net numbers to the wires connecting transistors by considering the contact cuts between the mask layers.

The *fourth step* identifies the net numbers of the terminals of each transistor, and determines the area of each transistor along with its channel type.

The *final step* does region analysis on the wires used to connect the transistors to obtain area and perimeter information for calculating capacitances.

The fundamental algorithm used by Goalie is the scanline algorithm [Lauther, 1981]. It sweeps a vertical line, called a scanline, across the layout. The layout is read in as a list of edge records sorted in order of their left endpoints. The scanline stops at each point where an edge begins, ends or intersects with another edge.

Goalie conducts the necessary processing by running up the scanline accumulating information about the edges crossing the scanline. It only needs to keep the edges that currently intersect with the scanline in memory; this is useful because the amount of main memory in most single processor computers is insufficient to store and process the entire layout for a large circuit.

For the second and third steps of the extraction, Goalie writes the output edges, with temporary net numbers, out to disk as the scanline passes their right end point. If it finds two edges with different net numbers belonging to the same net, it writes a record to disk marking a merge between the nets.

Figure 2.3 shows a case where this occurs. Goalie sorts the output edge records in order of their left endpoints so that it can use them as input for the next step of the extraction. As it does this, it assigns final net numbers to the edges. Szymanski and Van Wyk developed efficient techniques to achieve fast sorting and region numbering without requiring that all the output edges remain in memory.

Large circuits typically require the input and output of over 50 Megabytes of data. Goalie needs fast and efficient access to disk storage to prevent the I/O requirements of large circuits from seriously affecting its performance.

Researchers [Chiang, K-W et al., 1989] have made improvements to Goalie in GOALIE2. They have reduced the time taken to run up the scanline at each scanline stop by using a trapezoidal representation for polygons, which enables GOALIE2 to predict the range of y coordinates affected by a change in the status of an edge. This work involves the algorithms used in Goalie, but the techniques are easily transferable to GOALIE2.

This work makes use of the techniques used in Goalie that allow large circuits to be extracted by keeping most of the data on secondary storage. It allows multiple processor machines with a few nodes to extract large circuits without needing a large amount of expensive random access memory. It describes how to access secondary storage efficiently to prevent accesses from becoming a bottleneck. The techniques allow small scale multicomputers to achieve cost effective speedups compared to the serial machines currently in use.

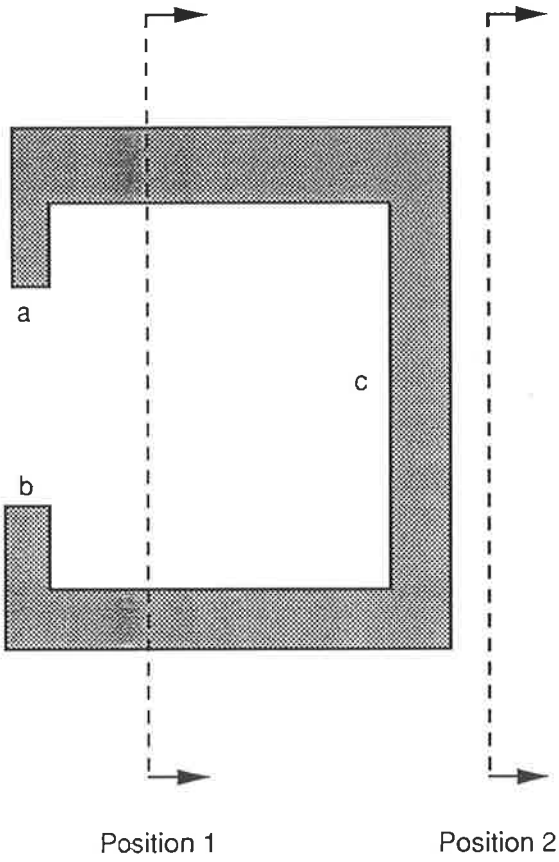


Figure 2.3: Example of a region merge

2.2.4 Strategy for Parallel Circuit Extraction

To show the feasibility of applying parallel processing techniques to the Goalie circuit extraction algorithms, this work will concentrate on the second step of the extraction. The second step involves the boolean combination of mask layers using a scanline algorithm, and applying unique numbers to the transistor regions. The third step of the extraction uses similar scanline techniques. The final two steps are simpler because they do not require a separate sorting and renumbering pass over the data.

The basic strategy is similar to the strategies of MACE [Levitin, 1986; Marantz, 1986] and PACE [Belkhale & Banerjee, 1988]. It involves splitting the flattened layout into sections, operating on these sections independently and in parallel, and combining the results in a merging stage. Figure 2.4 illustrates three strategies for partitioning the circuit. This work investigates different strategies for splitting up the layout, and the use of secondary storage to extract complete VLSI circuits.

MACE split the design up into equal size, horizontal slices. The number of slices corresponded to the number of processors available to do the extraction. By using horizontal slices, MACE minimized the amount of data required to be scanned at any scanline stop for a vertical scanline moving from left to right. The use of slices of equal size leads to load balancing problems among the processors handling the slices if the density of the circuit varies. MACE had problems merging transistors across slice boundaries. It ran in a distributed computing environment using a VAXcluster™ [Kronenberg et al., 1986]. It did not achieve any performance improvements over the serial circuit extractor that served as its basis, because of overheads in merging complete devices across boundaries.

PACE split the design into slices in a grid formation. The number of slices corresponded to the number of processors available in an Intel iPSC D4/MX

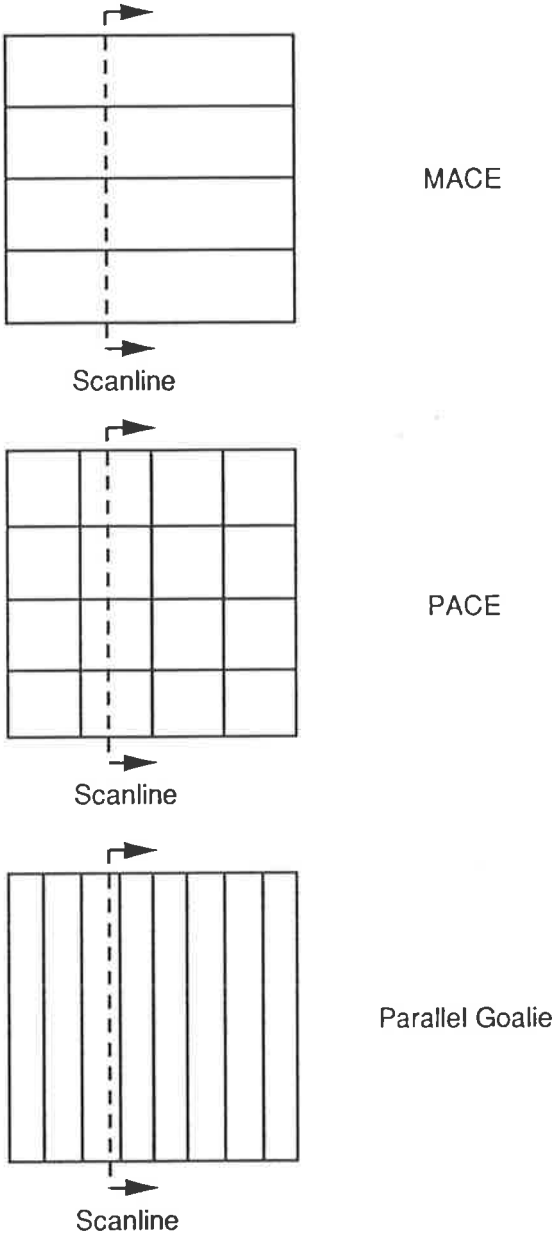


Figure 2.4: Partitioning strategies

hypercube. It selected the number of slices in the x and y direction to minimize the perimeter of the rectangular regions, and hence reduce the amount of work required to merge the results across slices. It used an efficient merging strategy, and achieved speedups of around 11-12 on a 16 processor hypercube compared to the running time on a single processor. The amount of main memory available on the processors in the hypercube limits the size of circuit that PACE can process. The largest circuit extracted contained 24,000 transistors. Load balancing would also cause a problem, as in MACE, if the density of the design was highly irregular.

Parallel Goalie split the circuit up into vertical slices to take advantage of the techniques used in Goalie that optimize the speed of processing the edges crossing a scanline. Even for large designs, it is feasible for processors to have enough memory to store the information associated with all the edges crossing a scanline. The amount of time to merge the results across the boundary is small enough compared to the time to process a slice that it is not worth adopting a grid based scheme to minimize the perimeter.

More recently Belkhale and Banerjee used a revised partitioning algorithm for PACE2, which incorporated a parallel algorithm for extracting the capacitances and resistances associated with the electrical nets and transistors [Belkhale & Banerjee, 1989]. They partitioned the circuit into vertical slices with equal numbers of mask rectangles, and further divided the slices into seqments containing equal numbers of rectangles. They achieved speedups of 13-14 for large circuits, and showed results with an improved load balance over their original partitioning method in PACE.

2.2.5 Parallel Goalie

2.2.5.1 General purpose multicomputer

Parallel Goalie, the parallel version of Goalie, uses the HPC/VORX local area multicomputer system as the general purpose, message based multiprocessor [Gaglianello et al., 1989]. HPC/VORX is a multiple instruction stream, multiple data stream (MIMD) architecture with distributed memory, which consists of clusters of communication links that provide low latency communications between processing nodes. Figure 2.5 illustrates this communications architecture.

The links, which have a bandwidth of 100 Mbits/sec, can either connect directly to a processing node, or to another cluster. They consisted of twisted pair ribbon cable, transmitting 8-bits with a 12.5 MHz clock to give a communications rate of 100 Mbits/sec over distances up to 150 feet.

The clusters consist of 12 ports that buffer and route messages, and they interconnect in a hypercube topology. As of October 1988, the system contained four SUN 3 workstations and 70 adjunct processors, with about 1.8 megabytes of memory available on each node, which is enough to support medium grain parallelism. The adjunct processors are Motorola 68020 based Multibus single board computers, some of which run at 16.67 MHz and others at 25MHz. It takes about $1ms$ to send a 1024 byte message using the software protocols provided by the VORX operating system [Gaglianello et al., 1989].

VORX, which is a direct descendent of Meglos [Gaglianello & Katseff, 1985; Gaglianello & Katseff, 1986], an operating system for the S/NET multicomputer system [Ahuja, 1983], provides a UNIX™ operating system programming interface. The SUN workstations (3/260s) act as host machines running SUN's version of the UNIX™ operating system. They provide software de-

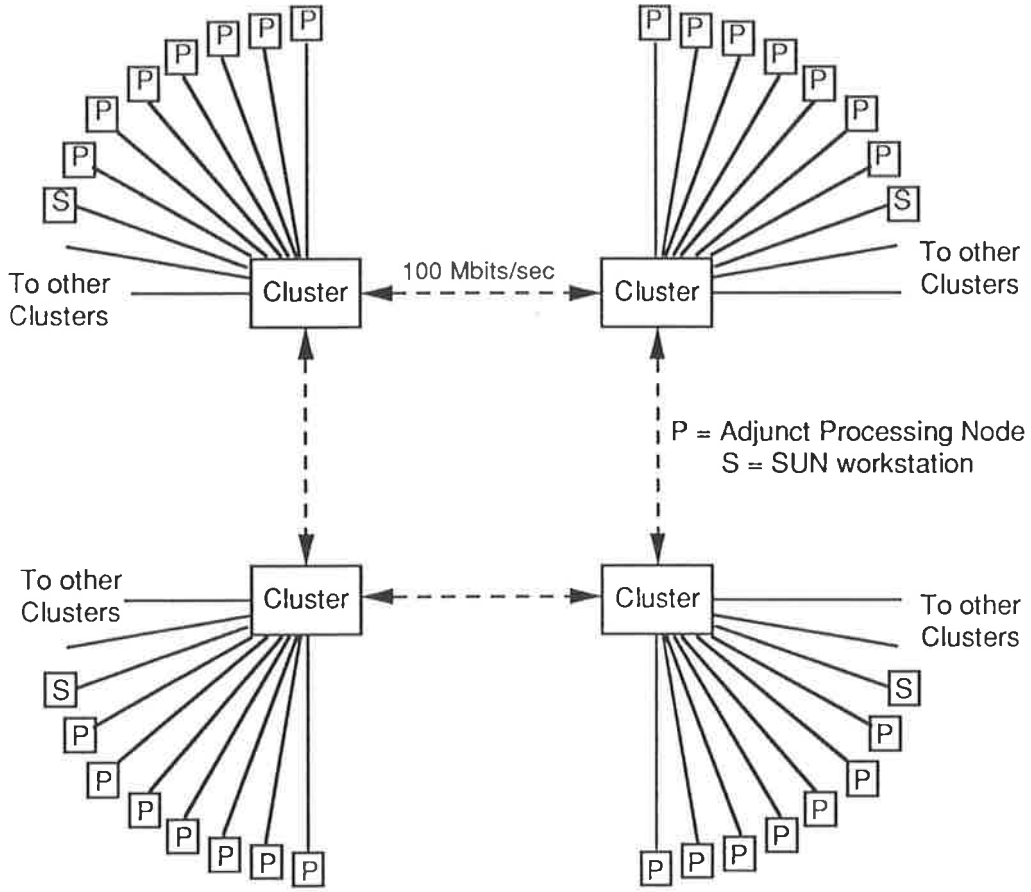


Figure 2.5: HPC Architecture

velopment facilities, and download code onto the adjunct processors. While carrying out this work, the only secondary storage available was through the SUN workstations.

Figure 2.6 shows the programmer's model of the HPC multicomputer. The software development facilities used in the development of Parallel Goalie include a C compiler, a symbolic debugger and a communications debugger. The C compiler interacts with the facilities of the HPC through C library functions. Programs communicate through channels which appear to the programmer like UNIX operating system files. There are library routines for opening, closing, reading and writing to these channels.

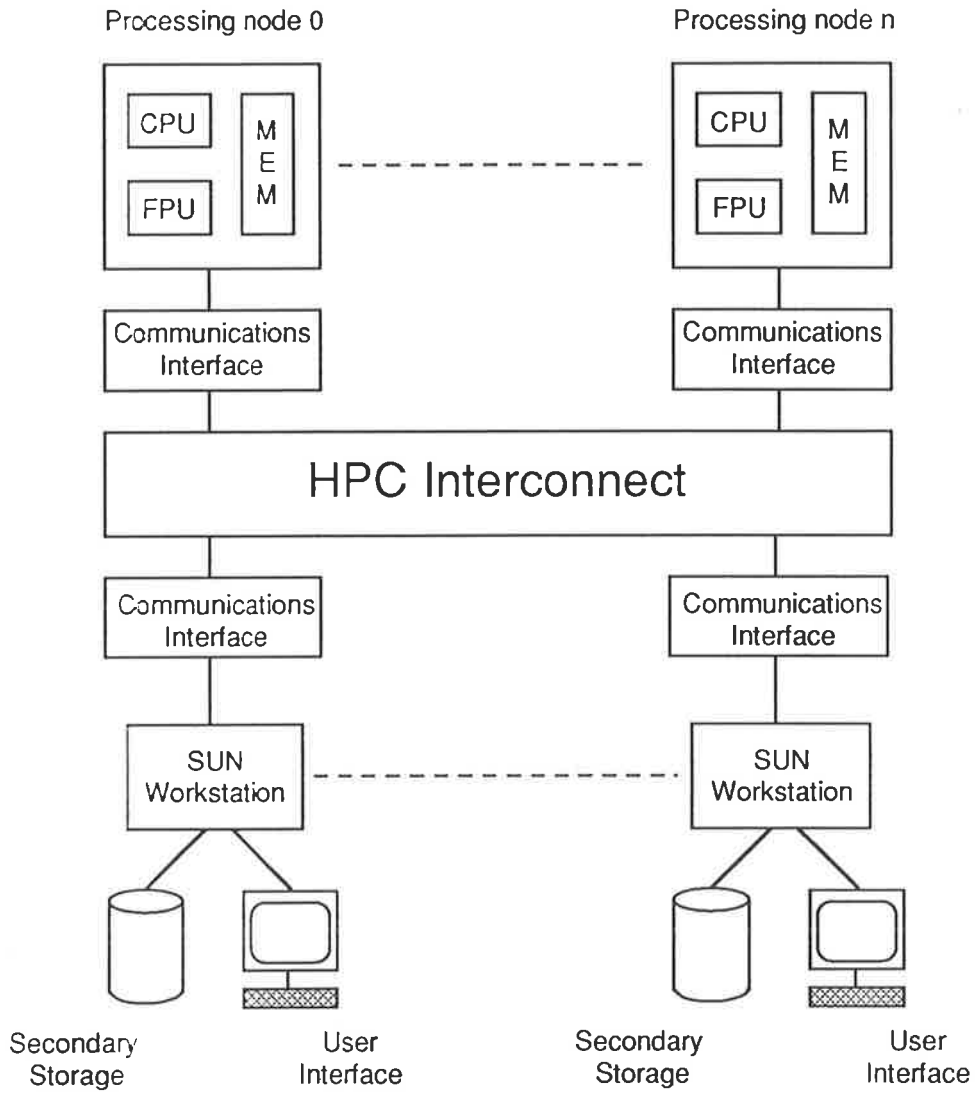


Figure 2.6: Programmer's Model of the HPC

2.2.5.2 Serial disk accesses

Parallel Goalie initially used a SUN workstation to flatten the hierarchical mask layout into the same set of edge files as the original Goalie. It divided each edge file into subfiles, one for each processor on the HPC participating in the extraction. These subfiles contained the edges for equal area slices of the layout. The SUN workstation serving as the host machine provided disk storage for these files. Parallel Goalie ran the section of the original Goalie code that created the intermediate layers on each of the participating processors. This test showed that the time taken to access files from a single disk limits the maximum speed up possible. When using one processor, Parallel Goalie spent about 30% of the running time in disk accesses. The maximum speed up was a factor of 3 without merging the results.

The load balance was particularly bad for layouts with an irregular distribution of mask geometry. This occurs often because even though most of the layout consists of blocks of circuitry that are regular, there is also a significant amount of area used for routing. The routing often concentrates in one area. The geometry in this area is usually sparse, and a processor assigned to this area finishes processing the geometry well before the other processors. This causes a load imbalance that results in a low efficiency for the number of processors used to solve the problem. To improve the load balance, Parallel Goalie now divides the layout into slices that contain almost equal numbers of edges.

2.2.5.3 Parallel disk accesses

To achieve significant speed ups, Parallel Goalie needed to improve the time spent in accessing disk storage. One option was to split the files across multiple hosts, instead of using a single host to process the requests from

the adjunct processing nodes, to allow Parallel Goalie to access disks in parallel. The problem with this technique is that the operating system on the workstations must handle the read and write requests. Parallel Goalie needs high speed, sequential access to large files; the workstation's file system is suboptimal for this task.

A more efficient technique is to use disks attached to the adjunct nodes, so that Parallel Goalie can optimize the file system on these disks. This assumes that Parallel Goalie has exclusive access to these disks while the program is running. The high production volumes of disks for personal computers has brought the costs of disks down. This makes it economical to provide separate disks for each processing node. Accessing a large contiguous block of data at a time is faster than incurring disk latencies to find several smaller blocks. Reducing the number of disk transactions also reduces the software overhead to begin the disk transfers. The optimum size of the block depends on the parameters of the system such as: the layout of the disk, the amount of main memory available for buffers on the processing node, the hardware available to control the disk transfers and the processing time for one block of data. If an I/O processor is available to manage the transfers from disk into main memory, the disk transfers will occur while the CPU is processing the previous block of data. Multiple buffers [Itai & Raz, 1988] for input and output ensure that the I/O processor and the CPU do not interfere with each other while transferring data. When data are being read from one large file and the results sent to another large file after a small amount of processing, which occurs in the sorting and region numbering phase of Goalie, two disk drives can reduce the number of seeks, and hence improve performance.

To show the advantage of using disks attached to the adjunct nodes, *spare adjunct nodes on the HPC were programmed to behave as storage nodes emulating disks.* Figure 2.7 shows this programming model. The storage nodes used their local memory to store blocks of data. A simple file server ran on

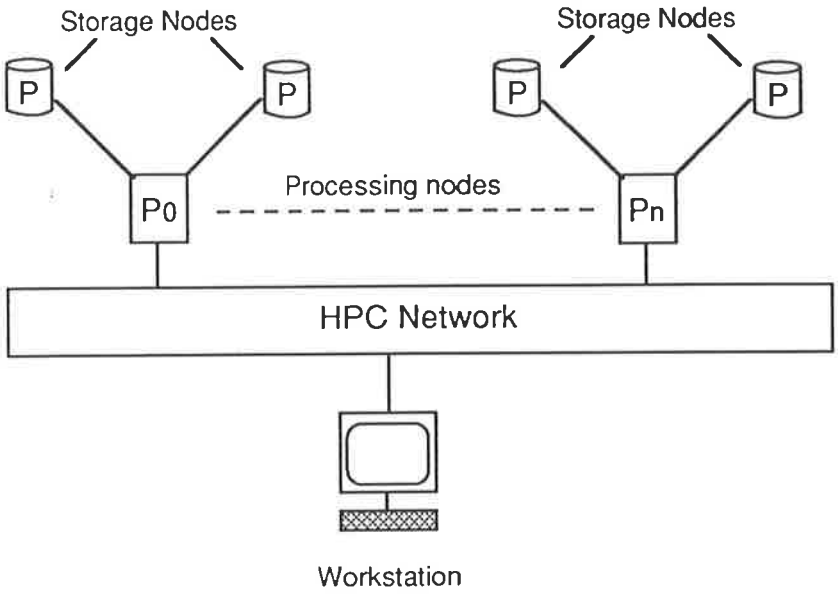


Figure 2.7: Programming Model used in Parallel Goalie

each of these nodes; it supported the standard UNIX™ operating system file operations such as open, read, write, unlink, and lseek. Each participating processing node had access to two storage nodes. One storage node supplied the input data, while the other node received the output data. This provided 3.5 Megabytes of secondary storage per processing node. The HPC has some nodes that run at 25 MHz, and some that run at 16 MHz. The 25 MHz nodes ran as processing nodes, while the 16 MHz nodes ran as storage nodes. There was about a $4ms$ latency in setting up the communications whenever accessing a block of storage from a storage node. The storage nodes provided a data transfer rate of 0.85 Mbytes/sec. A conventional disk drive should be able to provide higher performance.

2.2.5.4 Circuit partitioning

The first step in the extraction required processing the hierarchical mask layout and determining the slices to be sent to the processors. Parallel Goalie did this with the following steps:

- *Convert hierarchical description into a single data structure.* A SUN workstation read the hierarchical description from a file, and translated it into a single data structure kept in main memory. An adjunct processing node could not do this step, as the data structure was too large to store for typical designs.

- *Produce edge based flattened description.* Parallel Goalie traversed the data structure to produce an edge based, flattened description on a *single* file, in contrast to Goalie, which produces one file for each mask layer. It created a table containing a mapping between layer numbers, associated with each edge in the file, and layer names, provided by the user in the command line.

- *Sort edges.* Parallel Goalie sorted the edge based description file into a format suitable for use in the scanline algorithms.

- *Find slice boundaries.* By seeking into the file to locations corresponding to the end of equal size file partitions, where the number of partitions equals the number of processing nodes, Parallel Goalie found the x coordinates of the slice boundaries.

- *Send slices to storage nodes.* Parallel Goalie read the file sequentially, and wrote out the edges for each slice to the corresponding storage nodes. It split edges that cross slice boundaries, and stored the halves on separate storage nodes.

Researchers have found by measurement that the expected number of edges in a circuit design cut by any vertical line is $O(\sqrt{N})$, where N is the number of edges [Bentley et al., 1980]. Thus the expected number of new edges that Parallel Goalie creates at each slice boundary is also $O(\sqrt{N})$, which for large N will be a small percentage of N.

2.2.5.5 Layout analysis

Each processing node has a single file, stored in one of its storage nodes, containing the layout description for the slice assigned to the processor. Accessing a single file stored on disk is more efficient than having many separate files for each layer.

During the second step of the extraction, Parallel Goalie makes two passes over the data stored in secondary storage.

The *first pass* scans the layout, with the same scanline algorithm used in Goalie, to produce a file on its other storage node containing the edges describing the intermediate layers produced by the boolean combination of the input layers.

The *second pass* reads the edges from this file, sorts them in preparation for the third step of extraction, and assigns unique numbers to the transistor regions. It collects and stores the edges that touch the left and right boundaries of the slice in local memory. The output of the second pass goes to the storage node that stored the original edge file, which is no longer needed. The second pass takes about 25% of the total time for the second step of the extraction.

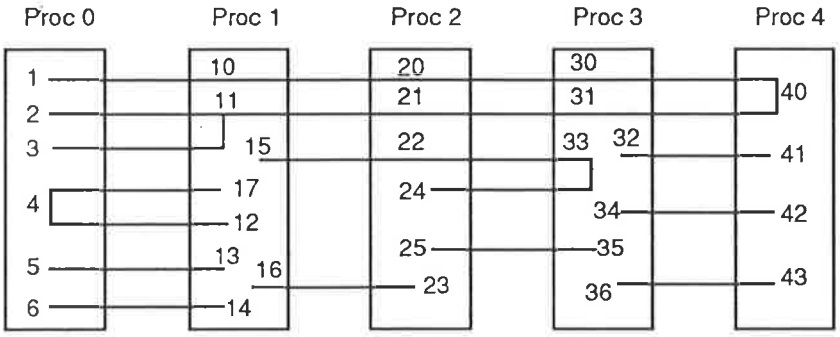
2.2.5.6 Merging region numbers

Parallel Goalie needs to merge the results from each of the slices before beginning the third step of the extraction. Each processing node assigns a unique series of numbers, derived from its node number, to each region. Thus the region numbers are unique across the design. The object of the merge is to detect regions that belong to the same unique region, and merge the region numbers associated with these regions. The merging phase takes advantage of most of the geometry being local to a slice. Most unique regions do not span more than two slices.

The following paragraphs describe the merging procedure. Figures 2.8 and 2.9 show an example of merging a set of wires split across 5 slices.

In the *first stage* of the merge, each processing node transmits the edges of its right boundary to the processor handling the slice on the right. The processor receiving the edges from the left slice compares them with its own edges touching that boundary. It produces a merge record for the region numbers of each matching pair of edges. It uses a hash table to index the merge records, and links records with the same hash key. Each merge record can link to a parent merge record. The parent record is the record containing the least region number associated with a region. At a later time Parallel Goalie can link the parent record to a new parent. This stage detects most of the region number merges, and creates an array of mapping records; each of which contains a child region number and its corresponding parent region number.

In the *second stage*, each processor with an array of mapping records sends the array to the processor handling the slice to the right. The processor receiving the array checks to see if it has any merge records, associated with its left boundary, with a child region number matching any of the ones contained in the mapping records. If it finds a match, it creates a merge



After Stage 1 of the Merge: (childnet -> parentnet)

10 -> 1	20 -> 10	30 -> 20	40 -> 30
11 -> 2	21 -> 11	31 -> 21	31 -> 30
3 -> 2	22 -> 15	33 -> 22	41 -> 32
12 -> 4	23 -> 16	24 -> 22	42 -> 34
13 ->5		35 -> 25	43 -> 36
14 -> 6			
17 -> 4			

After Stage 2 of the Merge

10 -> 1	20 -> 10	*30 -> 20	40 -> 30
11 -> 2	21 -> 11	*31 -> 21	*31 -> 30
*3 -> 2	22 -> 15	33 -> 22	41 -> 32
12 -> 4	23 -> 16	*24 -> 22	42 -> 34
13 ->5	*10 -> 1	35 -> 25	43 -> 36
14 -> 6	*11 -> 2	*20 -> 10	30 -> 20
17 -> 4		*21 -> 11	21 -> 20
		*22 -> 15	

* Placed in array of common mappings

Figure 2.8: Example of merging across slice boundaries

After Stage 3 of the Merge: (childnet -> parentnet)

10 -> 1	20 -> 10	30 -> 20	40 -> 30
11 -> 2	21 -> 11	31 -> 21	31 -> 30
3 -> 2	22 -> 15	33 -> 22	41 -> 32
12 -> 4	23 -> 16	24 -> 22	42 -> 34
13 ->5	10 -> 1	35 -> 25	43 -> 36
14 -> 6	11 -> 2	20 -> 10	30 -> 20
17 -> 4	3 -> 2	21 -> 11	21 -> 20
20 -> 1	24 -> 15	22 -> 15	3 -> 2
21 -> 2	31 -> 30	3 -> 2	10 -> 1
22 -> 15	30 -> 1	10 -> 1	11 -> 2
24 -> 22	2 -> 1	11 -> 2	20 -> 1
31 -> 30		2 -> 1	2 -> 1
30 -> 1			22 -> 15
2 -> 1			24 -> 15

After Stage 4 of the Merge:

2 -> 1	2 -> 1	2 -> 1	2 -> 1	2 -> 1
3 -> 1	3 -> 1	3 -> 1	3 -> 1	3 -> 1
	10 -> 1	10 -> 1	10 -> 1	10 -> 1
	11 -> 1	11 -> 1	11 -> 1	11 -> 1
	12 -> 4	20 -> 1	20 -> 1	20 -> 1
	13 ->5	21 -> 1	21 -> 1	21 -> 1
	14 -> 6	22 -> 15	22 -> 15	22 -> 15
	17 -> 4	23 -> 16	24 -> 15	24 - > 15
	20 -> 1	24 -> 15	30 -> 1	30 -> 1
	21 -> 1	30 -> 1	31 -> 1	31 -> 1
	22 -> 15	31 -> 1	33 -> 15	40 -> 1
	24 -> 15		35 -> 25	41 -> 32
	30 -> 1			42 -> 34
	31 -> 1			43 -> 36

Figure 2.9: Example of merging across slice boundaries (ctd.)

record for the parent region number in the mapping record, provided one does not already exist. It links the parent merge records corresponding to the matching child region numbers, according to which has the least region number. When it finds a match, it enters the mapping record into an array of common mappings. This array represents merges that have propagated across two boundaries. It also contains merges between two region numbers found to merge within the slice on the current processing node.

In the *third stage*, each processor with an array of common mappings sends the array to all the other processors that are processing a boundary between slices. Usually this array will be small so the communication overhead of this stage will be small. When each processor receives this array it creates new merge records for any region numbers that do not have a corresponding merge record. It links these records as for the second stage of the merge.

In the *final stage*, each processor that is processing a boundary finds the final parent record for each of its merge records by traversing the linked lists of records. The processing node handling the first boundary sends the merges for records to the processing node handling the first slice. Each processing node now has a collection of merge records indexed by a hash table. When the data stored on disk is read in for the next step of the circuit extraction, Parallel Goalie can use the data stored in the hash table to adjust the region numbers associated with the incoming edges. Alternatively, Parallel Goalie could store the merging information in an array and write it out to disk with the edges it refers to.

The merging procedure described above is applicable to both the second and third steps of the circuit extraction, which assign unique region numbers to transistors and the wires that interconnect them.

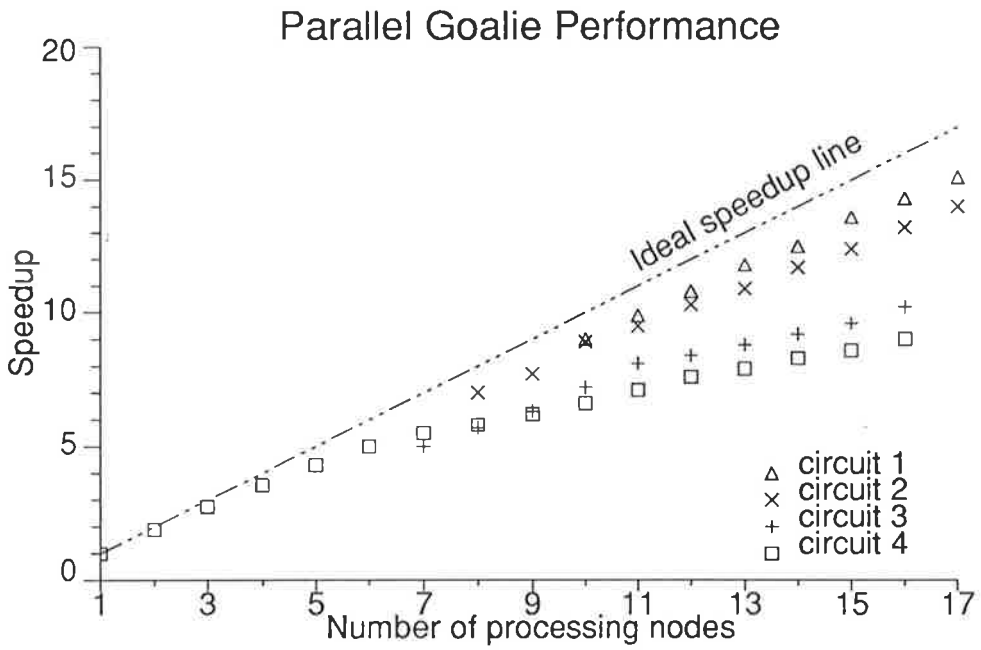


Figure 2.10: Performance results for the detection and numbering of transistor regions

2.2.6 Results

Figure 2.10 shows the performance of Parallel Goalie, for various MOS circuits described in Table 2.1, relative to the linear speedup line. The results are for the detection of diffusion wires and polysilicon wires, as well as the detection and numbering of transistor regions. The speedups are relative to the predicted processing time on one adjunct processing node, as there was insufficient storage capacity on the storage nodes to measure experimental results for a single processor. The processing time on a single node is within 1% of the time the original Goalie would take on a single node. Table 2.1 shows the size of the intermediate file produced from the first pass over the input data. The input edge records can describe rectangles by storing the bottom left and top right coordinates. Parallel Goalie splits rectangles into separate edges during the processing. The average number of edges intersect-

	circuit1	circuit2	circuit3	circuit4
Number of transistors	53700	35400	27450	900
Number of input edge records	340830	148600	157220	4940
Size of input file (kbytes)	8180	3567	3773	118
Size of intermediate file (kbytes)	18872	12324	11031	333
Number of scanline stops	17222	14960	14179	899
Average no. of edges in scanline	958	805	648	105
Maximum no. of edges in scanline	2351	1631	2064	294
Time for pass 1 on Sun 3/260 (secs)	470	259	247	7
Time for pass 2 on Sun 3/260 (secs)	161	106	91	3
Elapsed time on Sun 3/260 (secs)	631	365	338	10
Est. time for 1 processing node (secs)	950	560	400	10

Table 2.1: Characteristics of the circuits in Figure 2.10

ing a scanline agreed with the assumption that the number is proportional to the square root of the total number of edge records. The maximum number of edges that intersect with the scanline varied between 2 to 3 times the average. The position of the boundaries between slices can affect the speed of the merging stage. Table 2.1 includes the times for a SUN 3/260 executing the original Goalie for comparison.

Table 2.2 shows results for the slowest processing node when using 12 processing nodes. The total number of new edge records created by splitting up the circuit into 12 slices is small, relative to the total number of input edge records, for all the circuits except circuit 4. The times for the second pass over the data, which does the region numbering and sorting, are a smaller percentage of the total time compared to the results of the single processor SUN. This is because the second pass is more I/O intensive than the first

	circuit 1	circuit 2	circuit 3	circuit 4
Time for Pass1 (secs)	73	46	37	0.8
Time for Pass2 (secs)	13	6	7	0.3
Time for Merge (secs)	0.09	0.04	0.04	0.08
Number of messages	33	18	20	15
Total number of new edges	8805	5606	7048	811

Table 2.2: Times for the slowest node out of 12 nodes

pass, and the advantages of parallel disk accesses are more pronounced. The results show that the merging overhead is small. As the number of participating processing nodes increases, the number of new edges created will become of the same order as the original number of input edges. The overhead in accessing these edges on secondary storage will affect the efficiency. For small circuits such as circuit 4 there is little advantage in using multiple processors, as the extraction time on a single processor is small.

The efficiency of Parallel Goalie is dependent on the quality of the load balance. Circuit 2 and circuit 3 are about the same size, but show different performance characteristics. Circuit 3 has areas of dense circuitry and sparse areas of routing. Figure 2.11 shows the partitioning for circuit 3, and Figure 2.12 shows the variation in load among 12 participating processing nodes. The last two processing nodes have the most load and are responsible for the poorer performance of the program for circuit 3.

From Figure 2.13 the number of scanline stops in a partition appears to correlate with the processing time for a slice. Figures 2.14 to 2.16 show the results for circuit 2. These results also show a correlation between the number of scanline stops and the processing time for a slice. The variation in processing load is still small, so the technique of assigning equal numbers of

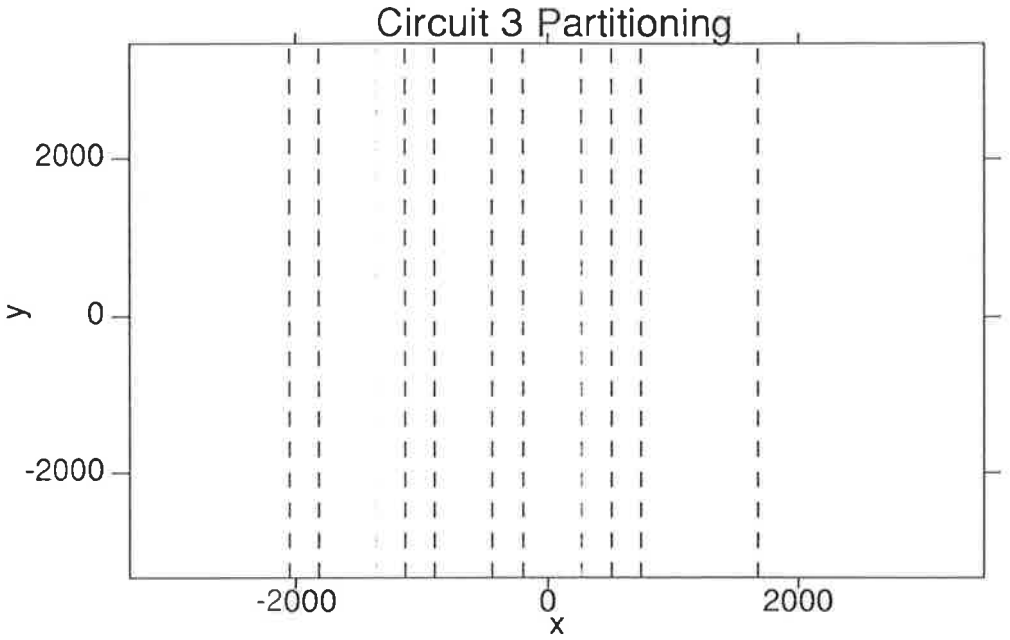


Figure 2.11: Partitioning for circuit 3 with 12 processors

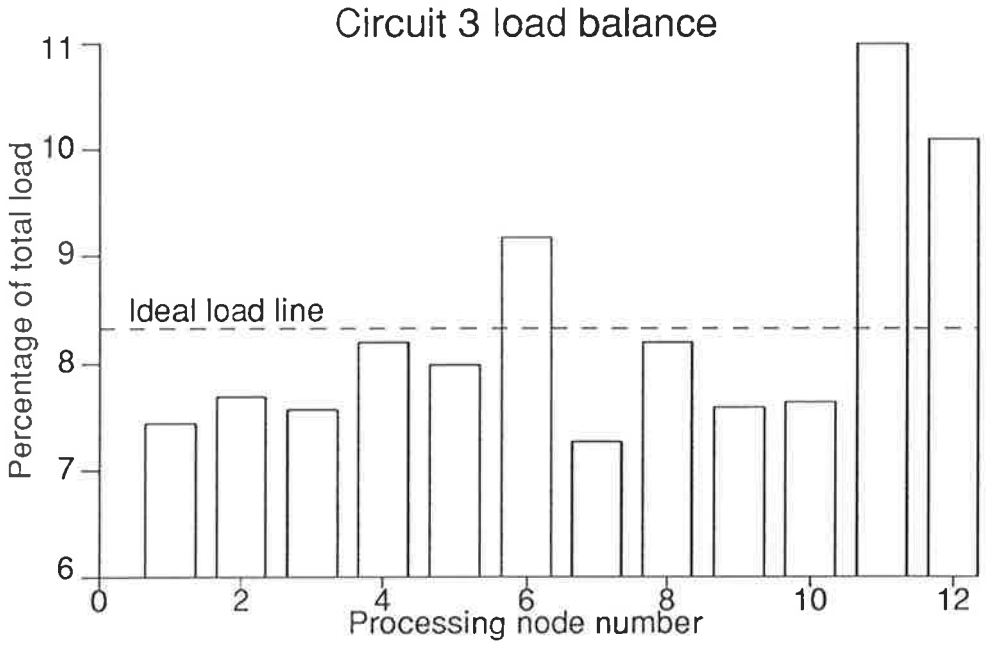


Figure 2.12: Load variation for circuit 3 with 12 processors

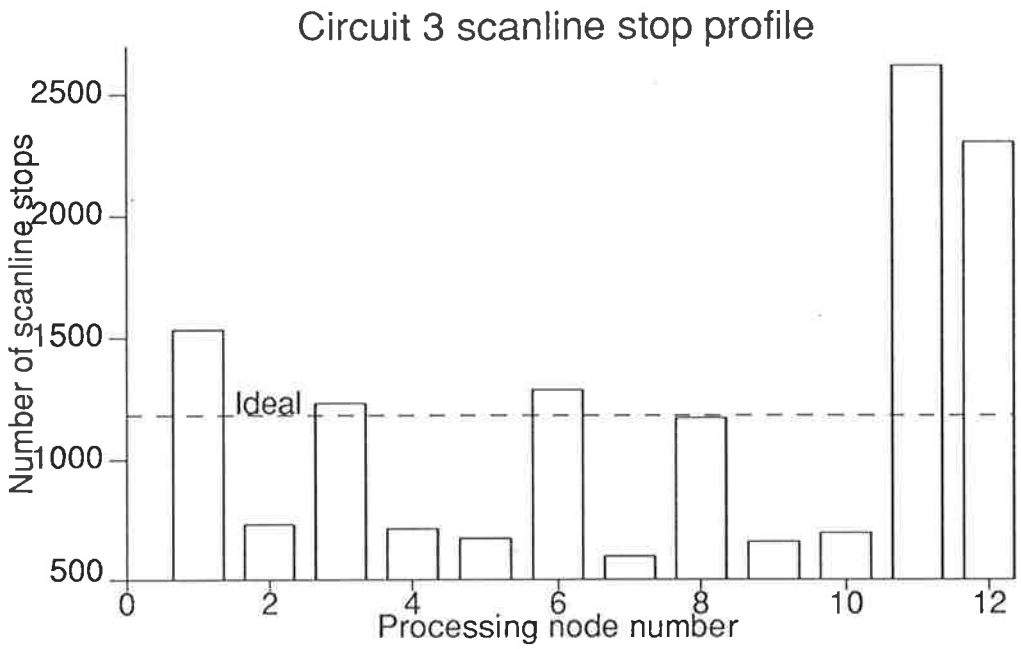


Figure 2.13: Number of scanline stops for circuit 3 with 12 processors

edges to each slice is effective. The results show that the software overhead in scheduling and processing the results of each scanline stop is significant, whereas the processing of the data in the scanline is efficient. Thus to further improve the load balance, Parallel Goalie could take into account the estimated number of scanline stops in a slice.

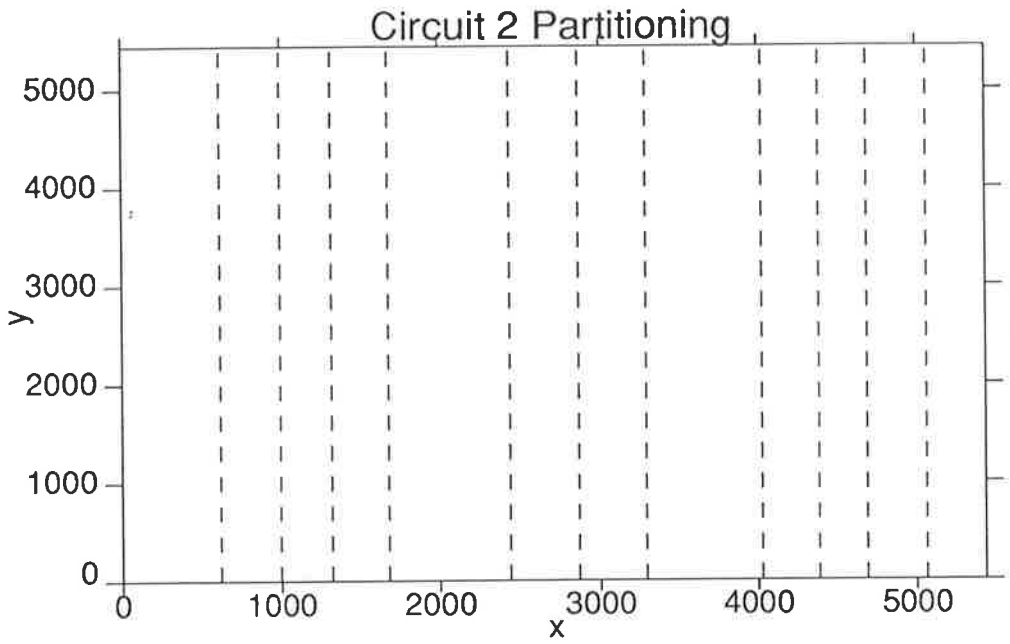


Figure 2.14: Partitioning for circuit 2 with 12 processors

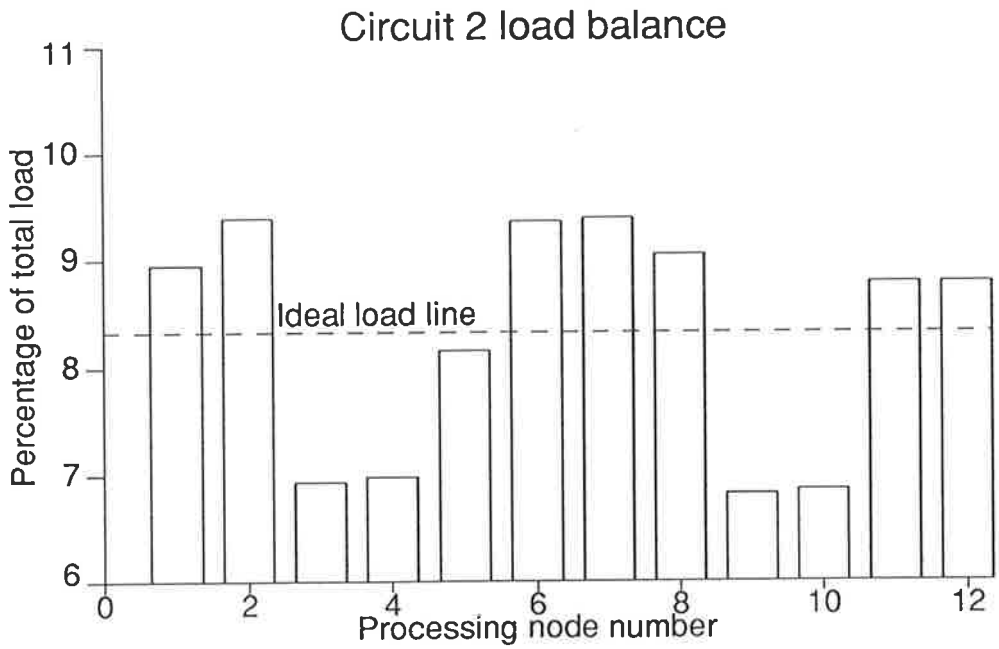


Figure 2.15: Load variation for circuit 2 with 12 processors

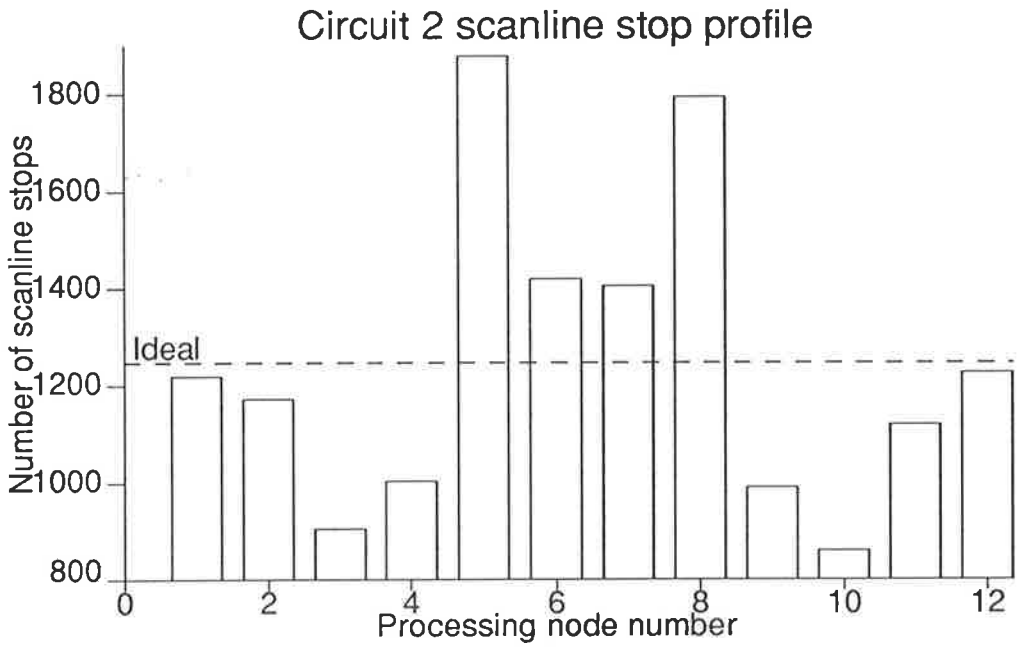


Figure 2.16: Number of scanline stops for circuit 2 with 12 processors

2.2.7 Conclusions

A multiple processor architecture must provide speedups for a range of compute-intensive tasks with differing characteristics, to be cost effective in meeting the needs of computer aided design. The minimization of execution time for only one task, such as circuit simulation, may result in another task becoming the design bottleneck.

The results from Parallel Goalie, which for large circuits achieved speedups of 15 when using 17 processing nodes, show that it is feasible to carry out circuit extraction on a general purpose multicomputer. The task of extracting a circuit from the flattened mask description of a complete circuit involves large quantities of data. To process this data, the multicomputer requires either a large amount of primary memory, or the use of secondary storage to hold intermediate results. The cost per megabyte of primary memory packaged on a board is typically more than one hundred times the cost of magnetic disk storage [Hennessy & Patterson, 1990, pp. 518,556]. Costs will limit the amount of main memory such that it is likely to always lag behind the increasing needs of applications that use a large amount of data. Thus secondary storage will always be necessary for these applications.

Parallel Goalie has simulated the use of secondary storage accessed in parallel to show how to achieve speedups on a general purpose multicomputer with parallel access to secondary storage. The techniques allow a small scale multicomputer with few nodes to achieve useful speedups in extracting a large circuit, provided there is enough secondary storage available. As processors become faster, disks for secondary storage will become more of a bottleneck. The use of disks dedicated for use by the multicomputer, which can optimize the file system on these disks for particular applications, allows the full use of the available disk bandwidth.

The efficiency of a multiple processor machine executing a parallel algorithm

is a measure of how close its speedups are to the ideal linear case where speedup equals the number of processing nodes. Parallel Goalie has achieved efficiencies as high as 88% for circuits with a regular mask geometry. Results indicate that the quality of the load balance governs the efficiency of the extraction,, rather than the communications overhead in merging the results. Thus an area for further research is finding the most efficient way to partition the circuit.

The development of Parallel Goalie has provided experience in using a multicomputer. This has led to the following recommendations for a high performance computing system, which consists of a general purpose computing section (GPCS) and a special purpose computing section (SPCS):

- use a multiple processor architecture,
- provide a high speed interconnection structure,
- use processing nodes with large amounts of local primary memory,
- provide parallel access to secondary storage,
- use a separate file system on the SPCS,
- use the GPCS to provide the user interface,
- provide a familiar software development environment,
- support conventional programming languages,
- provide debugging facilities to aid parallel programming,

Multiple processor computers are a cost effective way of providing speedup over uniprocessor general purpose computers. They are more flexible than special purpose accelerators, as they can execute a variety of different applications, and can take advantage of advances in algorithms. They also provide

faster communications, and are easier to load balance, than a distributed network of workstations.

Both the SPCS and the GPCS can take advantage of parallel processing. For the GPCS, which provides software development facilities, and support for the user interactive stages of design, small scale parallel processing is suitable. This is because most of the user interactive applications cannot make effective use of more than 10 processors. A small scale multiprocessor can consist of up to 10 processors interconnected with an industry standard bus. The use of shared memory will ease the development of a general purpose operating system. This simple architecture will soon be cost effective enough to use as a single user workstation. Although many applications will not use more than a few processors simultaneously, independent applications can run in parallel. For example a compiler may run on one processor while an editor runs on another.

For the SPCS, which provides the support for the compute intensive stages of design, medium scale (with between 10 and 100 processors) parallel processing is suitable. This is because compute intensive applications can use over 20 processors simultaneously. A machine with up to 100 processors allows several designers to execute applications on separate parts of the machine simultaneously. The processors can communicate using a sophisticated communications network. The use of distributed memory simplifies the design of the interconnection network by avoiding the shared memory problem of maintaining coherency between data stored in the local memory of different processors. A distributed memory multiprocessor communicating via messages suits the requirements of event driven circuit simulation algorithms.

High speed communications are essential to allow a multiple processor computer to efficiently execute parallel algorithms that require regular exchanges of information. For many parallel algorithms, the communications speed can

be the limiting factor in the speed up available from the multiprocessor.

The major goal in most parallel algorithms is to maximize the amount of work a processor can do with local data before needing to communicate with another node. This requires large amounts of local memory to allow processing nodes to maintain large data structures and keep intermediate results.

To allow the SPCS to process large amounts of data, without having to rely on the GPCS to provide secondary storage, the SPCS needs its own secondary storage system. Distributing this storage among the processing nodes of the SPCS allows parallel access to secondary storage. By using its own file system on this storage independently of the GPCS, the SPCS can optimize the available bandwidth to secondary storage.

By using the GPCS to provide a user interface for the SPCS, the system can use a simple operating system for the SPCS, while taking advantage of the sophisticated user-friendly facilities available on the GPCS. A familiar software development environment on the GPCS, such as the UNIX operating system provides, speeds the development of software for the SPCS. A system providing conventional programming languages, such as C, allows programmers to get immediate use from the system, and take advantage of many existing software development utilities. Although several parallel programming languages are available, none of them are widely used by programmers.

Programmers need additional debugging facilities to help develop parallel programs. Useful aids would include tools to show the activity and status of each of the processing nodes to detect communication problems.

2.2.8 Acknowledgements

I thank Bob Gaglianello, Howard Katseff, Beth Robinson, and Teri Lindstrom for their help and advice in using the HPC/VORX local area multicomputer system, Binay Sugla for his discussions on algorithms, Bruce Hillyer for his discussions on disk I/O, and Bob Clark for providing test circuits. Special thanks to Bryan Ackland for his advice and encouragement.

Chapter 3

Approaches to Accelerating Computer-Aided Design

This chapter will briefly review the different approaches to making faster computers, before it recommends using a Multiple Instruction stream/ Multiple Data (MIMD) stream approach for accelerating computer-aided design. It will then discuss the main characteristics of MIMD architectures. Finally it will analyse recent research into parallel CAD algorithms for running on MIMD architectures. On the basis of this analysis, the thesis will recommend using a heterogeneous multicomputer architecture for accelerating computer-aided design.

3.1 Introduction

To reduce the design time of complex circuits, computer-aided design tools need faster computers. As simulation can take many hours, ideally designers would like computers faster than current workstations by factors of a thousand or more, to allow interactive evaluation of circuit performance. By

reducing design time, computers will allow designers to take advantage of the benefits of high levels of integration for low production volume applications.

Faster computers are always in demand, because they provide

- more accurate results in a given time span,
- improved response time for interactive tasks,
- and allow more design iterations in a given time span.

For the same period of time, a fast computer can take calculations closer to convergence than a slower computer, thus providing more accurate results. High accuracy is important to designers who want a high degree of confidence that their computer simulations closely match the behaviour of the fabricated circuit.

For interactive tasks, a fast computer can bring the time to respond to a user's request closer to being instantaneous. This ensures that a designer can maintain concentration on the task at hand, hence improving productivity. Brady discusses how distractions can lower a designer's productivity [Brady, 1986].

Fast computers speed up the time to evaluate the quality of each design iteration. This encourages designers to experiment with more design options to provide a better chance of finding an optimal solution.

Thus a large amount of research has gone into developing fast computers. The next section will briefly review some of the different approaches to achieving high performance.

3.2 Strategies for High Performance

Researchers have come up with various taxonomies for classifying computers [Giloi, 1988; Hwang, 1987; Haynes et al., 1982; Johnson, 1988; Solchenbach & Trottenberg, 1988; Skillicorn, 1988; Kuck, 1982; Duncan, 1990]. This section follows the classification used in Helin and Kaski's report on the performance of high speed computers [Helin & Kaski, 1989]. They based their classification on Flynn's taxonomy.

Flynn [Flynn, 1966; Flynn, 1972] introduced the following simple terminology for classifying high performance computers:

- Single Instruction Stream - Single Data Stream (SISD)
- Single Instruction Stream - Multiple Data Stream (SIMD)
- Multiple Instruction Stream - Single Data Stream (MISD)
- Multiple Instruction Stream - Multiple Data Stream (MIMD)

A *stream* is the sequence of data or instructions that a machine sees as it executes a program. There is often disagreement among researchers about the class to which a particular computer design belongs.

Conventional uniprocessor computers, which execute one arithmetic operation for each arithmetic instruction, fall into the SISD category. Whereas array processors, which execute the same arithmetic operation on different elements of an array simultaneously, fall into the SIMD category. Finally multiprocessors that can execute independent streams of instructions on different data fall into the MIMD category. There are no typical examples of the MISD category.

The following sections discuss common types of high performance computer, organized according to the above classification. Some of these types may

appear under more than one of Flynn's classifications, but for convenience, this thesis discusses them in the context of only one of these classifications.

3.2.1 SISD

Most conventional general purpose computers are SISD machines. They often use pipelining techniques to improve performance. By breaking the task of executing an instruction into several subtasks, such as instruction fetching, instruction decoding, operand fetching, and operating on the operands, an SISD machine can execute a series of instructions like a production line. The design trade off is between a deep pipeline to overlap the execution of as many instructions as possible, and a shallow pipeline to reduce the time to refill the pipeline when there is a branch in the stream of instructions. Ramamoorthy and Li [Ramamoorthy & Li, 1977] give a thorough treatment of pipelining.

SISD machines use the fastest available circuit technology to achieve the highest performance. By increasing the execution speed of simple instructions, such as integer addition, most programs will run faster without software modifications. This protects the investment in existing software.

Modern microprocessors use high levels of integration to incorporate many of the features, such as pipelining, used by traditional SISD mainframes. There are two philosophies in microprocessor design: RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer).

3.2.1.1 RISC versus CISC

RISC (Reduced Instruction Set Computer) based microprocessors only execute simple instructions, such as loading a register from memory and adding the contents of two registers, rather than more complex instructions such as adding the contents of two memory locations. They execute most of these in-

structions in a single clock cycle. CISC (Complex Instruction Set Computer) based microprocessors execute many complex instructions, which often consist of a sequence of simple operations. These complex instructions support many addressing modes, and ease the task of writing high level language compilers.

CISC processors try to improve performance by using additional hardware to replace common sequences of instructions with a single complex instruction that takes less time to execute. Often these complex instructions are too general, however, and a compiler can replace them with several simpler instructions that carry out a particular action in less time. Thus RISC processors concentrate on improving the execution speed of the fundamental instructions by taking advantage of simpler circuitry. RISC processors rely on efficient compilers to take advantage of the simple instruction set. They use the circuit area saved by not supporting complex instructions, for additional registers to avoid going off-chip to access variables. Many high performance workstations are now using RISC technology. CISC processors are incorporating many of the features of RISC processors so that the distinctions between the two approaches are becoming blurred.

Gimarc and Milutinovic [Gimarc & Milutinovic, 1987] discuss the features of RISC processors in more detail. Stallings also presents a tutorial on the RISC approach [Stallings, 1988].

3.2.2 SIMD

There are several subcategories of processing architectures that fall into the SIMD category:

- processor arrays
  - array processors

  - associative array
  - systolic arrays
- vector processors
- multi-operation processors
  - VLIW (Very Long Instruction Word) processors
  - Dataflow processors

3.2.2.1 Processor arrays

Most processor arrays are designed for special purposes, such as image and signal processing, where the algorithms possess properties such as regularity, recursiveness, and locality, which a processor array can exploit [Kung et al., 1987].

Array processors consist of multiple arithmetic units executing the same operation on different data synchronously. A control unit broadcasts an instruction to all the processing elements. These elements may make minor modifications to the instruction, such as alter operand addresses, or they may ignore the instruction depending on the state of a flag bit. Array processors are suitable for well structured problems involving large arrays of data. The MPP (Massively Parallel Processor) and the Connection Machine are examples of such a machine [Batcher, 1980; Tucker & Robertson, 1988].

Associative array processors contain one processing unit for each word, or group of words, in memory. They operate on the memory as a whole, on a bit by bit basis, allowing them to search the entire memory simultaneously. They can select memory words by their contents rather than by their address, hence the term associative memory. This makes them suitable for database applications. The STARAN is an example of such a machine [Batcher, 1974];

it uses 256 bit words. The Connection Machine can also provide an associative memory function [Waltz, 1987].

Systolic arrays consist of arrays of special purpose arithmetic units. Data moves through these units in a single direction. Different sets of data can enter the array from different sides. Systolic arrays pump data in a way analogous to a heart beat; in one phase all the arithmetic units do a calculation, in the next phase they pass the result onto the next unit. They achieve speedup via the pipeline principle. Their interconnection network is optimal for the application. They are ideal for doing matrix multiplication and Fourier transformations, and can support high input data rates. They can provide a useful preprocessing function before passing data onto a more conventional general purpose computer.

Both SIMD and MIMD implementations of systolic arrays are possible. Matrix-1 is an example of an SIMD systolic array [Foulser & Schreiber, 1987]. Wavefront array processors work on the same principle as systolic arrays except they work asynchronously. Kung presents a good overview of the principle of systolic arrays [Kung, 1982]; Fortes and Wah give a more recent overview [Fortes & Wah, 1987].

3.2.2.2 Vector processors

Supercomputers make use of multiple pipelined functional units that can operate in parallel to achieve high speed beyond that possible with a SISD architecture. For example, the CRAY-1 has 12 functional units including separate units for vector addition, floating point addition for both vectors and scalars, scalar addition, and memory address addition [Russell, 1978].

A *vector* is an ordered set of elements, whereas a *scalar* is a single independent element. Vector computers minimize the control, decoding, and reconfigura-

tion overhead of executing many identical operations, such as vector addition and multiplication, by issuing a single *vector instruction.* This instruction sets up the functional units once, and specifies the start and end of the vector. Once a vector operation produces its first result, it produces subsequent results at the rate of one per clock cycle. The initial latency comes from the time taken to pass through the pipe of a functional unit. Chaining allows the result of one functional unit to be fed into another functional unit, instead of back to a register.

While a vector operation is in progress, a vector computer can initiate another instruction that uses unassigned functional units. In this way, a vector computer can execute multiple operations in parallel from a single instruction stream.

These computers are very efficient for applications that involve many repetitive operations with regular structures, such as vectors and matrices, but are not cost effective for applications that involve mainly scalar operations. This leads to Amdahl's famous law, which states that a machine will waste a major performance improvement in processing vectors, unless a similar performance improvement occurs executing scalar operations [Amdahl, 1967]. This is because most applications spend a significant percentage of their running time on non-vectorizable data management tasks. These data management tasks will dominate the running time, if only the vectorizable sections of code benefit from performance improvements.

3.2.2.3 Multi-operation

Multi-operation processors can issue multiple operations per cycle. This is distinct from vector processors, which can only issue one instruction per cycle, and achieve parallelism by issuing more instructions during the execution of the previous instruction. The goal is to take advantage of instruction level

parallelism, by using multiple functional units to process multiple instructions in parallel.

VLIW (Very Long Instruction Word) machines typically have instruction words hundreds of bits long to specify operations with multiple functional units. They need an efficient and complex compiler to rearrange program code to extract the maximum parallelism from the resources available. They are suitable for code that is not vectorizable but may have several instructions in an inner loop that can execute in parallel. Researchers have found that the average instruction-level parallelism in unoptimized code is about 2 [Jouppi & Wall, 1989; Smith et al., 1989]. The iWarp microprocessor is a recent example of this type of architecture [Cohn et al., 1989].

Jouppi and Wall define the term *superscalar* to describe a machine that can issue multiple instructions per cycle [Jouppi & Wall, 1989; Murakami et al., 1989]. The intention is to provide performance improvements for ordinary non-vectorizable code. The instruction decode unit of a superscalar machine looks at the incoming sequence of program instructions, and decides which instructions can execute in parallel. It does this by considering the availability of functional units, and the data dependencies between instructions. The concept differs from VLIW machines, where the multiple operations specifiable in one long instruction word do not exceed the resources of the machine. A superscalar machine selects the operations to perform in parallel at run time, whereas this is done at compile time for a VLIW processor. The potential performance improvements with superscalar machines are similar to those mentioned earlier for VLIW machines.

A recent review paper by Weiss discusses the above multi-operation architectures in more detail [Weiss, 1989].

Dataflow machines work on the principle of executing an instruction when its operands are available. They use multiple functional units to execute

these instructions in parallel. Their goal is to extract all the instruction parallelism available. Data flow graphs show the parallelism available in a program. Dennis describes the basic concepts of dataflow machines [Dennis, 1979]. A pure dataflow machine extracts the parallelism at run time. Other approaches rely on the programmer or the compiler to specify the order of execution. Dataflow is not cost effective where most of the parallel processing gain can come from exploiting data parallelism [Giloi, 1988]. The Cydra 5 supercomputer is a recent example of an SIMD implementation of a dataflow machine [Rau et al., 1989]. It uses a modified version of dataflow called directed-dataflow, and uses a Fortran compiler to allow existing FORTRAN source code to run on the machine.

3.2.3 MIMD

MIMD architectures can process multiple instruction streams simultaneously. They are the most flexible approach to high performance processing, and can simulate many of the architectures mentioned previously. They can exploit data parallelism by partitioning a problem into sections, and solving each section independently in the same way as array processors. Each processor operates independently, however, and can follow data dependent program branches. They can also use program (or functional) parallelism by partitioning a program into subprograms, and distributing these independent subprograms among the processors. They are not as fast or efficient as special purpose hardware based on array or pipeline processing principles, but offer the opportunity of accelerating the running time of a wide range of problems within one application area. Each processor is more complex than an equivalent processor in an SIMD architecture because of the need to do independent instruction fetches.

MIMD machines allow the use of a more cost effective circuit technology

(*e.g.* CMOS) to achieve the same level, or higher levels, of performance at less cost, compared to a conventional SISD machine using the fastest available circuit technology (*e.g.* ECL). Where cost is no object, they allow calculation rates above what is available from a uniprocessor approach using the fastest available technology.

3.2.4 Summary

The algorithms used in computer-aided design cover a wide range of computational techniques (see Chapter 1). Many of these algorithms do not have sufficient regularity to make use of array processors or vector processors. The most flexible environment that will support a wide range of design tools is the MIMD environment. Within each independent processor in an MIMD machine, techniques used in multi-operation architectures can take advantage of instruction level parallelism.

The next section will describe the characteristics of MIMD architectures in more detail. This will provide some background for understanding the terms used in the discussion on parallel CAD algorithms for MIMD machines in the following section.

3.3 Characteristics of MIMD architectures

The next sections will discuss the following parameters of MIMD architectures:

- circuit technology
- the two main subcategories of MIMD computers
  - Multiprocessor - shared memory

- Multicomputer - distributed memory

- number of processors - grain size
- type of processing node
- interconnection network

Several papers discuss the performance of commercially available MIMD machines [Gehringer et al., 1988; Wasserman et al., 1988; Helin & Kaski, 1989]. Dongarra brings together several papers on experimental parallel architectures [Dongarra, 1987].

3.3.1 Circuit Technology

This section will briefly review the main circuit technology options for building computers.

Giloi [Giloi, 1988] has identified the two main competing circuit technologies for integrated circuits for high performance computers as VHSIC (Very High Speed Integrated Circuit) and VLSI.

VHSIC technology has the following characteristics:

- ECL/GaAs fabrication,
- low to medium scale integrated circuits (up to 1000 transistors per circuit),
- high power consumption - requires expensive cooling,
- high packaging costs,
- high design cost,
- and 3 - 10 ns clock cycle time.

VLSI technology has the following characteristics:

- NMOS/CMOS fabrication,
- very large scale integrated circuits (100,000 to 1,000,000 transistors per chip),
- low power consumption,
- many standard components - decreasing design cost,
- high production volume components - low cost,
- compact packaging,
- and 30 - 100 ns clock cycle time.

Until recently, the fastest supercomputers have used VHSIC technology, along with vector processing, to achieve high performance. This is a *uniprocessor* approach: even though there are multiple functional units, these units cannot carry out independent processing. Successive machines have achieved higher speeds by increased clock rates and improved packaging. The limits in vector processing have forced manufacturers, such as CRAY, to use small scale (< 10 processors) parallel processing techniques to get the next advance in speed [August et al., 1989]. These machines still use expensive VHSIC technology, and use parallelizing compilers to improve the performance of existing software. The cost of these machines is normally beyond the budget of everyone except the largest government organizations and corporations.

The characteristics of VLSI technology have allowed the concept of microprocessor based, *personal* computers. As manufacturers have increased levels of integration and circuit speed, these computers now rival the early mainframes for raw integer processing power. Many new manufacturers are now attempting to take on the current VHSIC supercomputers, by using medium

to large scale parallel processing architectures based on cost effective VLSI technology, to provide the same performance at much less cost. Bell [Bell, 1985] discusses the advantages of microprocessor based multiprocessors. To compete with current VHSIC supercomputers they must achieve at least a factor of 10 speedup from the use of parallelism [Giloi, 1988]. This can be hard to achieve with some problems. As levels of integration increase, adding supercomputer features such as vector processing to mass produced VLSI processing nodes will become more cost effective.

Most circuit designers use single processor workstations based on VLSI technology for their computer-aided design requirements. This thesis proposes using parallelism in a cost effective way to provide performance improvements about 20 times the performance of a single processor workstation. To allow small design teams affordable access to this performance, this high performance computing system will use VLSI technology.

3.3.2 Multiprocessor

A Multiprocessor consists of multiple processors that execute independent streams of instructions, and share a common memory address space. If two or more processors access the same memory location, hardware and software protocols must arbitrate among the processors. This shared memory can become a bottleneck to performance [Stenström, 1988]. Often the processors use cache memory to reduce the frequency of main memory accesses, and provide processors with faster access to memory. The cost of providing high speed memory access and efficient arbitration limits the number of processors in this type of computer. A system with shared memory is easier to program, as processors can share variables and data structures. Mechanisms are necessary to coordinate access to shared variables. Dinning gives an overview of some of these synchronization techniques [Dinning, 1989].

Recent supercomputers such as the CRAY X-MP [August et al., 1989] and the Convex C2 [Jones, 1989] have used multiprocessor architectures with up to 4 processing nodes. The processors use high speed interconnection networks to the shared memory. Lower cost mini-supercomputers such as the Sequent Balance [Thakkar et al., 1988] use a shared bus to interconnect the processors and shared memory.

3.3.3 Multicomputer

A Multicomputer (also called a message based multiprocessor) consists of interconnected independent computers. Each computer has its own processor and memory, with an independent address space. Processors communicate by passing messages. Software developers arrange for each processor to have most of the information it needs, to minimize the number of messages. When there are large numbers of processors, each processor only connects directly to a subset of other processors, to reduce the number of connections needed. Messages between processors not directly connected pass through either intermediate processors or routing nodes. Tasks that require extensive intercommunication reside on neighbouring processors for efficiency. These computers have less hardware complexity than multiprocessors, allowing more processors, but they are harder to program. They are a loosely coupled architecture, in contrast to the tightly coupled multiprocessor architecture. Athas and Seitz discuss these computers in more detail [Athas & Seitz, 1988].

Commercial examples include the NCUBE [Hayes et al., 1986] and the Intel iPSC/2 computers: Ranka *et al.* discuss some of the considerations in programming such machines [Ranka et al., 1988]. The BBN Butterfly computer has a multicomputer architecture, but allows the processors to treat the memory on the other nodes as part of a single address space, as for a multiprocessor [Crowther et al., 1985; Waterman, 1987].

3.3.4 Grain Size

A parallel program consists of tasks that can run simultaneously. Stone [Stone, 1987] defines granularity as a measure of the size of an individual task on a parallel machine. Howe and Moxon [Howe & Moxon, 1987a] define it as a way of expressing the ratio of computation to communication in a parallel program. In *coarse-grained* parallelism, the ratio is high, and the individual tasks are large and require little communication. For example, these tasks may consist of several subroutines, and may be as large as a typical uniprocessor program. The tasks in *medium-grained* parallelism are smaller, for example simple loops of a subroutine. In *fine-grained* parallelism, the ratio is low, and the tasks are very small and simple. These tasks may consist of a single instruction, and have a high level of inter-communication. A software developer trades off between getting the highest degree of parallelism in a problem with fine granularity, which offers the potential of the highest speedup, and minimizing the amount of communication, to reduce the overall run time of a task.

The number of processors in a parallel computer determines the grain of parallelism available to application programs. The following classification shows the correlation between the grain of parallelism and the number of available processors:

- Very Coarse: 2 - 10 processors
- Coarse: 10 - 100 processors
- Medium: 100 - 1,000 processors
- Fine: 1,000 - 10,000 processors
- Very Fine: > 10,000 processors

Very coarse grained systems use a few (< 10) high performance complex processing nodes. These processing nodes may be recent 32-bit microprocessors incorporating floating point support and internal caches. They are often bus based, and execute tasks that easily divide into subtasks with little intercommunication. Coarse grained systems may still use a bus, but there may be several processors per bus module.

Fine grain systems use simple processing nodes, and need tasks that are capable of fine subdivision. These processing nodes usually use custom designed circuits, and can integrate several processing ndoes into one VLSI circuit. Few very fine systems are available. The Connection Machine, with an SIMD architecture, is the most notable example [Hillis, 1982; Tucker & Robertson, 1988; Hillis & Steele, 1986]. It uses 64,000 one bit processors that each run the same instruction stream on different pieces of data. The MasPar is a more recent example which can support up to 16,384 processors with 4-bit wide integer ALUs [Blank, 1990; Nickolls, 1990; Christy, 1990].

The finer the grain of the system, the harder it is to subdivide a task. At high levels of subdivision the subtasks usually need to communicate often. This implies that fine grain architectures need efficient communication networks.

Eager *et al.* discuss the trade offs between speedup and efficiency for the number of processors applied to a particular problem [Eager et al., 1989]. They conclude that the average parallelism available in a problem, which is the average number of processors busy during program execution assuming an unlimited number of processors, will help determine how many processors to use. Applications using too many processors will have low efficiencies (ratios of speedup to the number of processors), as some processors will be idle waiting for other processors, hence reducing the cost effectiveness of a particular implementation. Applications using a few processors will have high efficiencies, but will not achieve the maximum potential speedup, as

some calculations that could proceed in parallel will be waiting for a free processor.

Curves showing the speedup in the execution time versus the number of processors often show an asymptotic characteristic. Once the number of processors passes the knee in the curve, the speedup only increases marginally. A number of processors equal to the average parallelism will yield a speedup close to the knee of the curve.

3.3.5 Type of Processing Node

Coarse grained parallel machines tend to use complex, high performance processing nodes. Some machines use processing nodes that incorporate support for vector processing. Vector processing offers substantial performance gains for calculations that involve operations with large vectors and matrices. For application areas that cannot use this facility, it is better to allocate the cost of providing vector processing to providing faster or additional memory.

Some machines provide demand paged virtual memory support, at the cost of additional software and hardware complexity. Machines with this facility are usually coarse grained, shared memory, bus-based systems. Demand paged virtual memory allows applications to execute assuming there is more memory than is physically available. Secondary storage provides this illusion by storing parts of an application's memory space, and swapping these parts in and out of main memory when required by an application. Denning describes the concepts of virtual memory in detail [Denning, 1970].

Medium grained machines often use standard microprocessors to reduce the cost per processing node. Most machines have used CISC microprocessors with scalar floating point support. Now that RISC technology has matured, the next generation of machines may use microprocessors incorporating RISC

features.

Fine grained machines use very simple processing nodes that require custom design. The aim is to provide huge numbers of these nodes by taking advantage of a simple design, enabling multiple processing nodes per integrated circuit.

3.3.6 Interconnection Network

A parallel machine's interconnection network has a major influence on how the system will perform for particular applications. Many of the concepts of telephone switching apply to processor interconnection networks. A fully interconnected system using a crossbar switch guarantees that every processor can access any resource at any time. This scheme is expensive and the number of connections rise as N^2, where N is the number of nodes. Alternatives to this scheme include multistage switching networks (as for telephone switching), shared buses, and point to point connection schemes.

In point to point connection schemes, processors connect to a subset of the other processors. A network can arrange the topology of point to point connections to suit the communications structure of particular applications. Special purpose parallel machines often use structures such as two and three dimensional meshes, where calculations only involve nearest neighbour interactions. Other structures, such as hypercubes, try to reduce the maximum number of intermediate nodes that separate any two nodes in the network [Wiley, 1987; Hayes & Mudge, 1989].

The choice of connection network depends on the number of nodes, the type of application to run on the machine, and the cost.

Feng [Feng, 1981] provides a thorough survey of interconnection networks. Broomell and Heath [Broomell & Heath, 1983] discuss the categories and

historical development of circuit switching topologies. Seigel [Siegel, 1985] presents a study of interconnection networks for large-scale parallel processing in text book form.

Feng identifies four design decisions for a network:

- operation mode - asynchronous, synchronous or combined
- control strategy - centralized or distributed
- switching method - circuit, packet or integrated
- network topology - static, or dynamic

Synchronous operation provides for instruction or data broadcast. Asynchronous operation allows connection requests to be issued dynamically during the running of a program. Some networks combine these two approaches.

Centralized control uses a central controller to set up the network switching elements to route a message. Distributed control uses the individual switching elements to route the message according to information in the message header.

Circuit switching establishes a physical path between a source and destination before sending data. Packet switching sends data in packets that pass through intermediate switches, without ever establishing a complete physical connection between source and destination. It allows a source node to send a packet into the network, and let the network route the packet to the destination.

A static topology uses dedicated links between network nodes. Examples include mesh, hypercube and ring topologies. A dynamic topology uses active switches to dynamically configure the links between network nodes. Examples include multi-stage and crossbar networks.

Jesshope *et al.* review packet routing techniques [Jesshope et al., 1989], for processor networks including:

- store and forward,
- wormhole,
- virtual cut-through,
- and double-buffering.

Store and forward networks completely store each packet in an intermediate node before transmitting to the next node. This causes a high latency in transmission, which is proportional to the length of the packet and the number links traversed.

Wormhole routing reduces the latency of store and forward networks. It does this by advancing the head of an incoming packet to the next node, without waiting for the rest of the message to arrive, thus sending messages in a pipelined fashion. Nodes only need to buffer the smallest unit of a packet called a *flit*. If the header flit blocks, all flits in the network stop advancing, which ties up the communication links.

Virtual cut-through routing is similar to wormhole routing, but uses buffers to store messages when they block. This frees up the preceding communication links for other messages, hence increasing throughput.

Double-buffering routing uses buffers at the sender and receiver. If a message blocks, the sender will try to retransmit the whole message after a delay. The receiver buffers a message until the local processor is ready to process it.

3.4 Parallel processing for CAD

This section will review parallel processing algorithms for solving some of the compute intensive problems in the computer-aided design of integrated circuits. Most of these algorithms make use of general purpose MIMD parallel processing architectures. From a study of these algorithms, the thesis can determine the important characteristics needed by a MIMD architecture to support the design of integrated circuits.

The section is organized, in a similar way to the discussion of CAD algorithms in Chapter 1, as follows:

- Synthesis tools
- Static analysis tools
- Dynamic analysis tools

3.4.1 Synthesis Tools - Simulated Annealing

Researchers have designed several special purpose architectures to aid the time consuming steps of design synthesis, such as placement and routing. Blank [Blank, 1984] discusses several examples of these architectures, which mostly use pipeline and array processing techniques.

In the area of general purpose parallel processing, simulated annealing (Section 1.2.1.2) has attracted the most attention from researchers. It is an extremely time consuming method for optimizing designs, but can offer the highest quality solution; thus making it an ideal problem to show the advantages of parallel processing.

Kravitz and Rutenbar [Kravitz & Rutenbar, 1987] give an overview of the strategies for parallel simulated annealing in the context of standard cell

placement (Section 1.2.2.4). The two main approaches are

- move-decomposition,
- and parallel-move.

Move-decomposition works by decomposing the task of evaluating the cost function, after moving a cell or swapping the positions of two cells, into several subtasks that can run in parallel. Both functional and data parallelism are possible. In functional parallelism each processor performs a specific subtask, such as wire length evaluation, whereas in data parallelism each processor does all the calculations for a subset of the cells in a design. The decomposition of a task is either static or dynamic. Static decomposition allocates cells or functions to processors at initialization, whereas dynamic decomposition allows the allocation to vary in response to the load on each processor. The dynamic strategy suffers increased overhead during run time, but avoids load balancing problems. Move-decomposition has limited potential data parallelism, since each move only influences a small proportion of cells.

The *parallel-move* strategy executes multiple, complete moves in parallel. If each processor chooses moves independently, it is possible for the moves to affect each other. A processor may accept a move because the cost function is better locally, but the effect on the global solution may be worse. The two approaches are to allow interacting parallel moves, even though this will affect the convergence of the simulating annealing algorithm, or to prohibit the parallel acceptance of interacting moves. The first approach assumes that moving only a few cells out of many cells will have a negligible effect on the convergence properties of the algorithm. The second approach can either bias the initial selection of moves, to ensure processors only evaluate non-interacting moves, or filter out the effects of accepting interacting moves.

Kravitz and Rutenbar did experiments on a VAX™ 11/784 multiprocessor, consisting of four VAX™ 11/780 processors connected to 8-Mbytes of shared memory, running the MACH operating system. They stored the entire database for the placement problem in shared memory. By using static functional decomposition with 3 processors, they achieved a speedup over a serial algorithm of about 2. By splitting each move into up to 15 subtasks, and using a work queue to implement dynamic decomposition on up to 4 processors, they found that the scheduling overhead outweighed the advantages of dynamic scheduling. They developed a parallel move strategy that preserved the integrity of the simulated annealing algorithm by only accepting one move at once, and aborting any other move evaluations in progress at the time of accepting the move. This strategy is only effective at low temperatures in the annealing algorithm where the algorithm rejects most of the moves. Thus they proposed using a move decomposition strategy at high temperatures, and a parallel move decomposition at low temperatures. Their net strategy achieved speedups of about 2 with 4 processors. They concluded that with a large multiprocessor, clusters of processors could execute each move, and multiple clusters could execute several moves in parallel.

Durand gives an extensive overview of some of the other approaches used by researchers for standard cell placement [Durand, 1989].

Rose *et al.* used a multiprocessor consisting of six National Semiconductor 32016 microprocessors, each with 1 Mbyte of local memory, a common MULTIBUS backplane, and 1 Mbyte of global memory [Rose et al., 1988]. They replaced the high temperature part of the annealing algorithm with a heuristic, which used the processors in parallel to generate about 10 diverse, but plausible, solutions. Even with one processor, this method is about 1.5 times as fast as standard annealing. They chose the solution with the lowest cost function to anneal further at lower temperatures.

During the low temperature annealing, each processor had its own copy of the database containing the cell locations, and is responsible for a portion of the chip area. A processor can displace cells from its assigned area to another processor's area, but it can only exchange cells within its own area. If the number of cells in a processor's area drops below 75% of its original level, the algorithm will rebalance the cells among processors. Each processor performs its own moves asynchronously; when a processor accepts a move it transmits this information globally to the other processors. At low temperatures the number of move acceptances is low, hence reducing the amount of global communication needed. Errors occur between the time a processor moves a cell, and when it informs other processors of the move; if another processor makes a move during this time it is acting on inaccurate information.

Rose *et al.* found that their solution still converged to the same final cost function as regular annealing. They have predicted a total speedup over standard simulated annealing of between 10 and 13 using 10 processors. As Durand has concluded, the convergence of the annealing technique is resistant to small local errors in calculation.

Casotto *et al.* used an eight processor Sequent Balance [Thakkar et al., 1988], which is a shared memory multiprocessor [Casotto et al., 1987]. They stored the cell data base in shared memory, and used a single parallel annealing algorithm for all temperatures that uses processors to do moves asynchronously and in parallel. They dynamically vary the assignment of cells to processors to reduce the effects of errors. They achieve a speedup of about 6 using 8 processors.

Darema and Pfister [Darema & Pfister, 1987] used a six processor IBM 3090 multiprocessor to simulate the operation of their parallel simulated annealing algorithm on the experimental IBM RP3 (Research Parallel Processing Prototype) [G. F. Pfister et al., 1987]. The RP3 is an MIMD multiprocessor that

can accommodate up to 512 processing nodes. Each processing node has a 32-bit microprocessor from the IBM PC RT, vector and scalar floating point support, a 32 kilobyte high-speed cache memory, and up to 8 Mbytes of main storage. The entire memory system, with up to 4 Gbytes of main memory, is dynamically configurable into global and local sections during application execution. Accesses to any other processor's local memory, take a uniform 1.6 times as long as local references. This is a feature of the interconnection network that avoids applications having to take into account the physical location of data outside local memory. Darema and Pfister point out that this flexible memory system allows the machine to adapt to very different algorithm characteristics. Greater peak speed is possible with local computations and message passing, but access to large, complex data structures, and the even distribution of irregular amounts of work, are easier to achieve with shared memory.

Darema and Pfister store a single copy of the cell placement data in shared memory, and use multiple processors to do moves in parallel. Each processor randomly selects two cells, locks them, and evaluates the effect on the cost function of exchanging their locations. When a processor accepts a move it updates the shared database. Darema and Pfister evaluated several techniques to reduce the error in allowing moves simultaneously: the best technique synchronized the database each time it changed the temperature. Their simulation predicted speedups of 14 using 32 processors, but this was with only 81 cells; they expect better efficiencies for larger circuits.

Jones and Banerjee developed a parallel annealing algorithm for standard cell placement for a 64-node Intel Hypercube using the Intel iPSC Simulator running on a SUN 3/50 workstation [Jones & Banerjee, 1987]. The Intel Hypercube is a distributed memory multicomputer that uses message passing to communicate between processors interconnected in a hypercube topology. They assigned an equal area portion of the chip to each processor,

and initially assigned cells randomly to different processors. At each time step, pairs of processors participate in a move operation. A master processor can exchange cells, or displace a cell, within its own area; or it can exchange cells with, or displace a cell to, a slave processor. Between iterations, the algorithm alternates a processor's role as slave or master. When processors accept a move, they send the move information to other processors to update local databases. Jones and Banerjee have estimated a speedup of between 11 and 21, using a 64-node machine, over a uniprocessor version of the algorithm.

More recent work by Sargent and Banerjee improved on the work of Jones and Banerjee for the Intel hypercube [Sargent & Banerjee, 1989; Banerjee et al., 1990]. They used a different partitioning scheme that allocated to each processor in the hypercube an entire row or set of rows of the standard cell layout. This scheme localized the computations needed to calculate the cost function. The simulated annealing algorithm used an adaptive cooling schedule based on the characteristics of the cost distribution and the annealing curve. Sargent and Banerjee included techniques to control the error associated with allowing several parallel moves to occur in sequence before doing a global cell position update. Sargent and Banerjee reported that the resulting algorithm performed better than the Jones and Banerjee algorithm, and experimental results showed close to linear speedups on the hypercube for large circuits.

Durand [Durand, 1989] reported work by Jayaraman and Rutenbar [Jayaraman & Rutenbar, 1987] who have developed an algorithm for floorplanning on the Intel Hypercube. They used a combination of move decomposition and parallel moves to achieve speedups between 4 and 7.5 with 16 processors.

Durand [Durand, 1989] also reported research on parallel simulated annealing algorithms on the massively parallel, SIMD, Connection Machine us-

ing 16,000 processors [Casotto & Sangiovanni-Vincentelli, 1987; Wong & Fiebrich, 1987]. Sequences of adjacent processors store data structures for each cell and net. Each processor is responsible for maintaining its portion of the data structure. At each iteration several hundred cell data structures consider a move simultaneously. The researchers reported that their algorithm converged despite the errors induced by simultaneous moves.

Kling and Banerjee developed a new algorithm called *Simulated Evolution*, which is analogous to the process of natural selection in a biological environment, for standard cell placement [Kling & Banerjee, 1989; Kling & Banerjee, 1990]. The algorithm determines a goodness value, which is high when neighbouring cells cluster close together and indicates the survival chance of a cell in its present location, for each cell in a placement. The algorithm removes all cells with a goodness below a random number from the placement, and then constructively re-inserts these cells into the placement to generate the next placement in the iteration. Periodically the algorithm randomly changes, or *mutates*, the placement to avoid falling into local minima.

Kling and Banerjee implemented a parallel version of the Simulated Evolution algorithm on a network of Sun workstations. A master processor sends the locations of all the cells on a placement grid to slave processors at the beginning of each iteration. Each slave processor executes the simulated evolution algorithm on a subset of the cells, and transfers the new placement back to the master processor ready for the next iteration. Kling and Banerjee reported speedups close to 4 using 4 workstations for large circuits.

Brouwer and Banerjee implemented a parallel simulated annealing algorithm on an Intel Hypercube with 16 processors to solve the channel routing problem [Brouwer & Banerjee, 1988]. Each processor was responsible for an equal number of routing tracks in a routing channel. The algorithm distributed the nets to route among the processors. The processors pair up to allow net

exchanges and net displacements in a similar way to Jones and Banerjee's standard cell placement technique. During the operation of the algorithm, connections between nets can overlap over the same routing track; the objective is to reduce this overlap to zero. Brouwer and Banerjee reported near linear speedups for large circuits.

Zargham implemented a parallel channel routing algorithm on a Sequent Balance 8000 (shared memory multiprocessor) using up to 6 processors [Zargham, 1988]. The results for one example showed a speedup of 2.6 with no improvement beyond 4 processors.

Brouwer and Banerjee have recently implemented a parallel implementation of a hierarchical global router, on an Encore Multimax (an eight processor, shared memory multiprocessor), that produces high quality routes [Brouwer & Banerjee, 1990]. They achieved speedups greater than 6 for up to 8 processors, and predict that their algorithm is scalable to larger numbers of processors.

Rose evaluated several techniques for speeding up global routing on the Encore MULTIMAX [Rose, 1988]. The two most promising methods were distributing the task of routing complete multi-terminal wires among processors, and distributing the task of evaluating the alternative routes of a two terminal segment. Both methods used a shared cost array that contained, for each routing position in a channel, the number of wires passing horizontally and the cost of traversing a row of circuit elements into the next channel. Processors update this array as they lay or rip up wires. Rose achieved speedups of 7.6 with 8 processors for large circuits with wire-based parallelism, and 4.6 using 8 processors with route-based parallelism. Rose stated that the two techniques can be combined when more processors are available.

3.4.1.1 Summary

The most time consuming computation steps in circuit synthesis are in the optimization tasks such as placement and routing. Optimizing a design can lead to substantial gains in performance and yield.

Simulated annealing provides high quality solutions, but with a high computation cost. Researchers have implemented parallel algorithms on both shared memory multiprocessors, and distributed memory multicomputers. The most common technique is to allow processors to evaluate changes to the design in parallel. The machines with shared memory tend to keep a single copy of the current cell layout in shared memory, whereas distributed memory machines tend to distribute this information among processors, because no single processor typically can accommodate the entire design. Results quoted for shared memory machines are typically for up to 10 processors, where good processor utilization ratios are evident. Distributed memory implementations quote results for up to 64 processors, but suffer from low utilization ratios for high numbers of processors, presumably because of the communications overheads of keeping other processors informed of design decisions. It would be useful to see results for algorithms implemented on tightly coupled shared memory machines with up to 100 processors, to see how the algorithms scale with the number of processors.

The best architecture solution, as Darema and Pfister pointed out, appears to be to support both shared and local memory. This supports the use of large shared data structures, and allows processors to have their own local memory for storing their own copy of program code and for local computations.

So far parallel algorithms for simulated annealing on MIMD processors have shown maximum levels of concurrency of about 20 for large circuits. The overall computation time will depend heavily on the power of the individual processors. Clearly, for this level of available parallelism, a parallel machine

must consist of processors that each at least match the performance of current uniprocessor workstations, to provide a performance advantage over a modern uniprocessor workstation.

Techniques other than simulated annealing have shown promising results for routing.

For other design synthesis tasks, such as interactive design input. individual cell generation, and compaction, which are not as compute-intensive, a modern uniprocessor workstation is appropriate.

3.4.2 Static Analysis Tools

3.4.2.1 Design rule checking

Geometrical design rule checking is amenable to parallel processing because of the locality of the design rules, allowing multiple processors to check different areas of a circuit independently. The main function of design rule checkers is checking the clearance between mask polygons, which is a local operation. This is in contrast to the task of circuit extraction, where a single net, such as a power net, may extend over large areas of the circuit.

Researchers have investigated several special purpose approaches to speeding up this time consuming task. Macomber and Mott discussed the architecture of the FAST-MASK Engine for layout verification, which uses Motorola 68020 based processing boards (each with 2 megabytes of local memory) that share a common bus called QBUS [Macomber & Mott, 1985]. Each processor checks different pieces of a design simultaneously.

Blank *et al.* described an array architecture with single bit processors for processing mask layouts represented as bit maps [Blank et al.. 1981]. Rutenbar *et al.* gave a good overview of array processing architectures for layout

analysis, and described a class of architectures called *raster pipeline subarrays* that consist of a pipeline of subarray stages [Rutenbar et al., 1984]. These subarrays are small arrays, such as 3x3, that act like a window on a bit map image, which feeds into the pipeline in a serial stream (raster order). They can identify a pattern in a bit map image of a mask layout that corresponds to a design rule violation [Seiler, 1982].

Kane and Sahni discussed an edge-based systolic array architecture for design rule checking [Kane & Sahni, 1987]. Carlson and Rutenbar discuss the design of hardware to accelerate the scanline maintenance and scanline processing stages of edge-based layout analysis [Carlson & Rutenbar, 1987]. Their scanline processing hardware design makes use of pipeline processing techniques.

Researchers have also developed design rule checkers for general purpose parallel machines. Gregoretti and Segall used the Cm* multiprocessor system, containing 50 processors with access to shared memory, to implement their design rule checker [Gregoretti & Segall, 1984; Gregoretti & Segall, 1987]. The Cm* multiprocessor consists of clusters of processor/memory modules, where each processor has its own local memory, and can address the local memory of other processors [Swan et al., 1977]. There are buses for both interconnecting modules in a cluster and interconnecting clusters. Gregoretti and Segall make use of the hierarchical description of a circuit, and use shared memory to store both the description of the circuit and the task queue. Initially the algorithm places a task on the queue for finding the intersections between elements in each cell of the design. Processors take tasks from the queue and execute them simultaneously. When a processor finds an intersection between elements of a cell involving a subcell, it creates a new task on the queue. Gregoretti and Segall achieved nearly ideal speedups for highly regular circuits, using up to 16 processors, but the performance dropped off for irregular circuits where a few large leaf cells dominated the running time.

Bier and Pleszkun made use of the locality in design rule checking by identifying the maximum design rule interaction distance (DRID) [Bier & Pleszkun, 1985]. They divide a flattened circuit description into partitions such that each partition overlaps its neighbours by at least the DRID. Multiple processors can process each of these partitions independently. A post processing step removes the design rule violations found in the overlap area of each partition. The variation in the processing time for each of these partitions limits the potential speedup of using multiple processors because of load balancing problems. Bier and Pleszkun predicted a potential speedup of 8 for a large circuit that they checked on a uniprocessor.

Carlson and Rutenbar developed algorithms for design rule checking on the massively parallel, SIMD Connection Machine [Carlson & Rutenbar, 1988]. Their algorithm assigns a separate edge of a flat, edge-based layout to each processor. If the number of edges exceeds the number of physical processors, the operating system can partition the physical memory of each node into segments to allow virtual processors to share a physical processor. Each processor is responsible for operations involving its edge, and can communicate with processors managing neighbouring edges. Carlson and Rutenbar put together a benchmark of tasks needed for layout verification, including sorting, boolean combination of mask layers and region numbering, to show the potential speedups available. They achieved speedups of between 89 and 244 (compared to a serial version of the benchmark running on a Vax 11/785 running BSD4.2 UNIX) for a 16,000 one bit processor machine, ignoring the time to load the data onto the machine. Note however, that the serial version uses file I/O between each operation, because of limited main memory, so the speedups don't necessarily represent the level of parallelism achieved by the algorithm.

Carlson and Rutenbar extended the work to produce a complete mask checking system called JIGSAW [Carlson & Rutenbar, 1990]. They compared the

performance of their program on the Connection machine with the DRACULA package from Cadence Design Systems running on an Apollo DN4000 workstation. They showed speedups of 112, using parallel I/O and 64,000 processors, for an industrial circuit with 76,000 edges.

3.4.2.2 Circuit extraction

Researchers have made use of general purpose parallel machines for circuit extraction, using similar techniques to those used in design rule checking [Levitin. 1986; Belkhale & Banerjee, 1988; Belkhale & Banerjee, 1989; Carlson & Rutenbar, 1988: Tonkin, 1990]. Section 2.2 discusses this in more detail.

3.4.2.3 Test generation

Fujiwara and Motohara used a 64 node multicomputer called LINKS-1 to develop a parallel test generation algorithm [Fujiwara & Motohara, 1988]. The processing nodes contained a Z8000 CPU with 1 MB of memory. The parallel algorithm interleaved test generation with fault simulation. The strategy involved allocating different faults to separate processors, and allowing each processor to generate a test vector for its fault. Other processors apply fault simulation for each generated test vector to determine which other faults the vector can detect. Both fault simulation and test generation can occur in parallel. Fujiwara and Motohara programmed the processors to act as either test generators or fault simulators; a single processor acted as the fault table manager and allocated tasks to the other processors. They obtained speedups of 20 using 40 processors. To avoid bottlenecks at the fault table manager processor they suggested using independent groups of processors with their own fault manager. each handling a subset of all the faults.

Fujiwara and Inoue analysed a parallel algorithm for distributed test simulation that allocated clusters of faults to networked server computers, which perform test generation and fault simulation [Fujiwara & Inoue, 1989]. They investigated the optimum size of the clusters taking into account varying communication overheads.

Banerjee [Banerjee, 1988] discussed methods for parallel test simulation and referred to previous work [Kramer, 1983; Chandra & Patel, 1988].

Patil and Banerjee give a good overview of parallel test generation techniques [Patil & Banerjee, 1990]. They mention fault parallelism, heuristic parallelism, simulation parallelism, AND parallelism and OR parallelism:

- *Fault parallelism* divides the fault set among processors [Fujiwara & Motohara, 1988]. The use of independent fault sets reduces the chance of a processor deriving a test vector that could also be used for a fault assigned to another processor, and gives good speedup. The faults that are hard-to-detect with a uniprocessor algorithm will still cause long computation times, because of the large number of backtracks necessary. Typically the algorithm aborts generating a test after exceeding a backtrack limit.

- *Heuristic parallelism* involves using several different heuristics (up to 5 or 6) in parallel to find a test vector for a fault, hence taking advantage of the best heuristic for a particular case.

- *Simulation parallelism* partitions the circuit among the processors, and uses parallel logic simulation techniques to speed up the forward implication step of test generation, which checks for conflicts between the primary input values. This will reduce the time to evaluate backtracks, improving the chances of finding a test for hard-to-detect faults. The level of circuit activity in test generation can be low, however, limiting

the amount of parallelism.

- *AND parallelism* involves dividing test generation into subtasks, which must *all* be completed, in a divide and conquer approach. Multiple processors can execute these subtasks in parallel, but if there is a large amount of shared data, the access overheads will limit the speedup.

- *OR parallelism* refers to evaluating in parallel different choice points of the decision tree used to find a test vector. Searching disjoint portions of the search space concurrently, leads to high processor utilization ratios as the amount of interprocessor communication is low.

Patil and Banerjee implemented a parallel version of the branch and bound search of the input space of a circuit used in the PODEM algorithm. When a conflict occurs in the choice of circuit inputs, the algorithm must backtrack through the choices made in previously assigned inputs. Multiple processors can evaluate in parallel alternative choices for the previous assignments to find the choice which led to the conflict. The algorithm ensures that processors are searching disjoint search spaces. Patil and Banerjee implemented the algorithm on an Intel iPSC/2 hypercube, and used a single processor to act as a dynamic scheduler. The communications requirements are low, which suits the distributed memory hypercube. For the same total backtrack limit, the parallel algorithm has a much better chance of finding a test vector for a hard-to-detect fault than the uniprocessor algorithm. Patil and Banerjee quoted near linear speedups, over a uniprocessor algorithm with the same net backtrack limit, using up to 16 processors.

3.4.2.4 Summary

Techniques for both design rule checking and circuit extraction mainly involve partitioning the circuit into sections, and operating on these sections

independently. Similar techniques should apply to electrical rule checking. The main limitation to performance is the regularity of the circuit. Near ideal speedups are possible with highly regular circuits, but the algorithms show much poorer performance for irregular circuits.

Researchers have used both shared memory and distributed memory machines. Distributed memory machines seem more appropriate for this form of data partitioning, which involves mainly local operations with low communications overhead. The high efficiencies achieved with regular circuits offers hope for effective use of larger numbers (e.g 50 - 100) of conventional processors, with a cheaper interconnection network than for shared memory machines.

Further research into better partitioning schemes may allow more effective use of larger numbers of processors, when circuits are irregular.

Test generation can make use of many processors for large designs, by partitioning the faults to be detected among the processors. Each processor can generate tests for its set of faults. Once a processor finds a test, it can use another processor to find which other faults the test will detect, using fault simulation. Again a distributed memory arrangement seems appropriate because of the localized computations involved.

Timing analysis could benefit from parallel processing, and further work is necessary in this area. Techniques similar to those used in parallel logic simulation may be applicable.

3.4.3 Dynamic Analysis Tools

3.4.3.1 Circuit simulation

Saleh *et al.* gave a good overview of the parallel approaches to both direct and relaxation methods of circuit simulation, with an emphasis on the use of shared memory multiprocessors [Saleh et al., 1989]. They concluded that techniques that improve the amount of parallelism in an algorithm come at the expense of more computation (which may negate the effect of increased parallelism), increased storage, and increased overhead. Frequently a software developer must modify an algorithm to fully exploit the performance of a particular computer architecture, applying special attention to the data access scheme.

In the direct method of circuit simulation, the most time consuming steps are forming the matrix of the set of coupled linear equations and solving this sparse matrix. For large circuits, the solution step dominates the execution time. Multiprocessors can speed up the matrix forming step by evaluating the models of individual devices in parallel, and updating the shared matrix data structure [Cox et al., 1986]. Solving the sparse matrix in parallel is more difficult. Multiprocessors can exploit fine-grain parallelism in a single pivot of LU decomposition, or they can carry out several independent pivot operations in parallel, with possibly limited parallelism. Other methods include *node tearing* techniques that partition the circuit into subcircuits connected by interconnection nodes. Multiple processors can evaluate the subcircuits in parallel with the disadvantage of additional matrix terms [Cox et al., 1986; Yeh & Rao, 1988].

Chang and Hajj reported speedups of 7 using an 8 processor Alliant FX/8 for direct method simulation [Chang & Hajj, 1988]. Sadayappan and Visvanathan discussed the use of shared-memory vector multiprocessing, and

showed speedups between 6 and 7 over a single scalar processor, using 6 vector processors [Sadayappan & Visvanathan, 1988]. Yang has shown speedups of 4.5 (for circuits up to 330 transistors) over the SPICE2 direct method simulator on an 8 processor Alliant FX/8 [Yang, 1990]. Yang reports that the speedup improves as circuits become larger generating more parallel tasks.

Lewis described the Awsim-3 hardware accelerator for circuit simulation, which consists of a special purpose device evaluator, and a programmable general purpose processor [Lewis, 1988]. The general purpose processor uses a VLIW architecture, which can execute up to 5 instructions per cycle, with a pipeline 13 clock cycles long.

Deutsch and Newton discussed the MSPLICE program, which uses the Iterated Timing Analysis relaxation method [Deutsch & Newton, 1984; Waterman, 1987]. They partition circuits into subcircuits for parallel evaluation, and can use direct methods to solve tightly coupled subcircuits. They reported speedups of 7 using a 10 processor BBN Butterfly multiprocessor, making use of local memory for nodes internal to a subcircuit, and shared memory for nodes referenced by multiple processors.

Jacob *et al.* carried out further work with MSPLICE using up to 99 processors on the BBN Butterfly [Jacob et al., 1986]. They identified bottlenecks resulting from contention for shared data structures as the number of processors increased. They state that effort must go into reducing the size of dynamically varying global data structures, which the algorithm cannot simply copy and store in the local memory of processors, as well as reducing the number of references to these structures. Their results show near linear speedups up to about 25 processors, with the speedup levelling off at about 20.

Saleh *et al.* reported results for parallel implementations of iterated timing analysis and waveform relaxation algorithms showing speedups between 4

and 5 on an 8 processor Alliant FX/8 [Saleh et al., 1989]. They used a simulator called PARASITE to predict the performance of their algorithm for larger numbers of processors. Their predictions show speedup levelling off at about 10 for their test circuits.

Hung *et al.* discussed relaxation techniques that exploit the hierarchical memory organization of the Cedar machine [Hung et al., 1990]. The Cedar machine consists of several shared memory Alliant FX/80 machines connected to a shared global memory. Thus the machine has two levels of shared memory. The strategy was to divide a circuit into large subcircuits, and distribute them among the Alliant machines. These machines cooperate to solve the circuit with parallel waveform relaxation techniques, using the shared global memory to pass the waveforms between subcircuits. Each Alliant machine uses parallel processing internally to solve its subcircuits.

Hung *et al.* compared the performance of parallel implementations of direct method, iterated timing analysis (PSLICE), and waveform relaxation (PRELAX) algorithms. They concluded that direct methods performed best for very tightly coupled subcircuits, PSLICE performed best for large moderately coupled subcircuits, and PRELAX performed best for large loosely coupled subcircuits (making the method suitable for using at the Cedar level of the machine). Their solution was to allow the Alliant machines to dynamically choose between using parallel direct method or PSLICE at run time, based on statistics gathered during the circuit partitioning process. Their results showed a speedup of about 9 using 4 Alliant machines with 4 processors each.

Deutsch *et al.* discussed a commercial software package called PowerSPICE, which runs on a Sequent multiprocessor with hardware acceleration modules (SPICEngines), that uses both direct and parallel relaxation methods [Deutsch et al., 1986].

Odent *et al.* discussed a multiprocessor version of the CSWAN program, which uses the waveform relaxation method [Odent et al., 1989; Dumlugöl et al., 1987]. They discuss techniques for partitioning feedback loops into small subcircuits, so that they can simulate several subcircuits in parallel using direct methods. This contrasts with previous approaches to partition circuits which result in large feedback loops assigned to one subcircuit. The simulation of the large subcircuit can dominate the simulation time. Odent *et al.* iterate the simulation of the subcircuits until the node waveforms in the feedback loop converge. They use dataflow techniques to schedule the simulation of subcircuits so as to achieve good processing-load balancing. In addition, they use parallel element evaluation and time-segment pipelining. Parallel element evaluation speeds up the formation of the matrix used in the direct method simulation of subcircuits. *Time-segment pipelining* involves dividing the simulation interval into time-segments and scheduling subcircuits for evaluation over a time-interval when its inputs are available over the same time-interval. Odent *et al.* achieved near linear speedups for large circuits, using up to 8 processors on a shared memory Sequent Balance. Parallel element evaluation was more useful for small circuits, where the number of subcircuits that the algorithm could execute in parallel was small.

Webber and Sangiovanni-Vincentelli discussed a parallel relaxation algorithm for the massively parallel Connection Machine [Webber & Sangiovanni-Vincentelli, 1987]. The algorithm uses Gauss-Jacobi relaxation with point decomposition, where the algorithm decouples each equation from the others, in contrast to block methods where equations group into subsystems. It maps each data item to a unique processor, and uses pointers to other processors to indicate the interconnections between devices and nodes. The Connection Machine allows a programmer to specify the number of bits of floating point resolution needed for each variable, hence providing a trade off between execution speed and memory usage, and the level of precision. The

disadvantage of Gauss-Jacobi is it can take a long time to converge for some circuits. The running time of the algorithm is independent of the circuit size, but a 65,536 processor Connection Machine limits the largest circuit to about 10.000 nodes. Webber and Sangiovanni-Vincentelli quoted results for a 630 node EPROM circuit, compared to both a waveform relaxation and direct method simulator running on a MicroVax under Ultrix, with speedups of 8 and 30 respectively.

3.4.3.2 Timing simulation

Lewis described both a uniprocessor (Awsim-1) and a multiprocessor (Awsim-2) hardware accelerator for timing simulation (Awsim-1,Awsim-2) [Lewis, 1988]. They both use hardware table look up techniques for device evaluation, and use a forward Euler integration algorithm with a small time step. The multiprocessor accelerator partitions a circuit into subcircuits with arbitrary collections of nodes, allowing each processor to simulate a separate subcircuit in parallel. When the simulation of a subcircuit requires knowledge of the node voltages from a subcircuit on another processor, the algorithm replicates the voltages of the shared nodes on each sharing processor. At the end of each time step, each processor managing a shared node broadcasts its updated value to the other processors sharing that node. This method relies on the calculation time at a time step being much larger than the interprocessor communication time. Allowing arbitrary collections of nodes in a subcircuit gives better control of load balance, but there is still a trade off with communication bandwidth between subcircuits. Lewis designed the hardware accelerator with 32 special purpose processors, and simulated a speedup of 19 for a 13,317 transistor test circuit.

Ackland *et al.* used a message based multiprocessor (or multicomputer) to implement a parallel version of a MOS timing simulator [Ackland et al.,

1985; Ackland et al., 1986] . The parallel algorithm generates a data flow graph for the circuit description, which shows the dependencies and potential concurrency [Ashok et al., 1985a; Ashok et al., 1985b]. It uses a forward integration scheme that reduces the time step whenever a node voltage changes by more than a preset threshold, or increases the time step when a node is relatively inactive. This method ensures accuracy and stability of the integration scheme, and takes advantage of circuit inactivity. The algorithm partitions the circuit into regions containing tightly coupled nodes, which have a bidirectional constant current source, such as a pass transistor, joining them together. The regions connect to one another using uni-directional signals; this maps well onto a message passing architecture. The algorithm sets the time step for each region independently, and uses a separate process to simulate each region as inputs become available. Each processor handles several regions, and uses buffering for messages between regions to allow source nodes to proceed ahead of their destination nodes. When a process simulating a region runs out of input data, or it has a blocked output channel, a processor suspends the process and resumes a process for a different region. Ackland *et al.* tested the algorithm on a M68000 based, 12 processor multicomputer and achieved speedups of about 5 with 7 processors. The disadvantage of this method is that some regions can be much larger than others, causing load balancing problems [Lewis, 1988].

3.4.3.3 Logic simulation

Most logic simulators (see Section 1.4.2.3) use a form of event-driven simulation (see Section 1.4.1.3) to take advantage of low levels of activity in a circuit. Smith [Smith, II, 1986] gives a thorough treatment of the design issues for implementing a parallel logic simulator. Franklin *et al.* [Franklin et al., 1984] also describe parallel logic simulation techniques.

Blank [Blank, 1984] and Franklin *et al.* [Franklin et al., 1984] survey several hardware simulators including HAL [Takasaki et al., 1987] and the Zycad logic evaluator [van Brunt, 1983]. These accelerators use multiple, special purpose, pipelined units to achieve high performance. Ashok *et al.* describe the possibility of using a special purpose dataflow machine for event-driven simulation [Ashok et al., 1985a; Ashok et al., 1985b].

Kravtiz *et al.* developed a parallel implementation of the COSMOS (COmpiled Simulator for MOS) switch level simulator for the Connection Machine [Kravitz et al., 1989a; Kravitz et al., 1989b; Bryant, 1988]. COSMOS breaks the circuit into subnetworks, and generates a boolean description for each subnetwork. The parallel algorithm divides the boolean operations for a subnetwork into logical ranks, and assigns a processor for each operation in a rank. The time to evaluate a module is proportional to the number of ranks. The algorithm schedules the modules for execution taking into account the number of available processors. The module with the most ranks determines the limit to the parallelism. In contrast to most parallel logic simulation techniques, the algorithm does not use an event driven model: it evaluates the entire circuit model each simulation step. Low levels of circuit activity, however, cause most of the evaluations to be redundant. Kravitz *et al.* point out that in the presence of massively parallel resources this is not a concern. They have achieved levels of parallelism in the thousands, but have not compared the running time with a serial algorithm on a conventional processor.

Chung *et al.* discussed an implementation of the asynchronous time-warp [Jefferson, 1985] event driven algorithm for gate level simulation on the Connection Machine [Chung & Chung, 1989]. They assigned a separate processor for each gate, and dynamically assigned the management of each event to a separate processor. They also did not compare results with the running time of a similar algorithm on a conventional processor.

The MARS (Microprogrammable Accelerator for Rapid Simulations) accelerator consists of multiple units called clusters [Agrawal et al., 1987; Agrawal & Dally, 1990]. Each cluster handles a partition of a circuit, and contains 14 special purpose processing elements interconnected by a crossbar switch, which are microprogrammable to form a multistage pipeline for a particular algorithm. In addition a cluster contains a M68020 based housekeeping processor for loading circuits, managing local disk storage for circuit partitions, and handling exceptions. The designers initially designed the accelerator for event-driven logic simulation, where the pipeline stages may handle steps such as event scheduling, fanout updating, or logic device evaluation. This simulator is an attempt to provide a special purpose architecture that is programmable to execute several different CAD tasks. However, it does not have the flexibility and ease of programming of a general purpose multiprocessor.

The key to maximizing the performance from a multiple processor machine that simulates separate partitions of a circuit in parallel is the partitioning scheme. An optimal partioning scheme maximizes the concurrency by keeping all processors busy at each time step, and minimizes the interprocessor communication. Finding this optimal partition is an NP-Complete problem, and could require more time than the simulation itself, hence heuristics are necessary for this task. Several researchers have examined the concurrency available in test logic circuits, and evaluated the effects of different partitioning schemes on architectures with different communication costs [Agrawal, 1986; Frank, 1986; Briner et al., 1988; Chamberlain & Franklin, 1988; Smith et al., 1988; Mueller-Thuns et al., 1989].

Static partitioning heuristics include:

- random,
- trace,

- and minimum distance partitioning [Deutsch & Newton, 1984].

A *random* partitioning heuristic randomly allocates circuit nodes to processors with an even distribution.

A *trace* partitioning heuristic allocates nodes, connected in a serial path, to the same processor; if more than one fanout element is present, it allocates the other fanout nodes to different processors.

A *minimum distance* heuristic allocates adjacent, non-local nodes, such as other fanout nodes, to adjacent processors in a communication network where the interprocessor communication time varies with the processor location.

The overhead of dynamic partitioning to balance the load can be excessive, because of either movement of the circuit description data, or additional remote memory references.

Smith *et al.* ran computer simulations on 11 test circuits for different message based (local memory only) parallel computer architecture models with both 4 and 16 processor configurations [Smith et al., 1988]. They tested varying ratios of logic element evaluation time to message transmission time, and three different partitioning heuristics, with a separate partition for each processor. One of the heuristics involved adding a circuit element to the partition with the most fan-in connections, while another worked with the most fan-out connections. These heuristics sought to group connected devices in the same partition to minimize interprocessor communications. The third heuristic used random partitioning. Smith *et al.* quoted results as an average for the 11 test circuits. They found that with ideal communications the random partitioning gave the best speedup with values of 3.5 for 4 partitions, and 10.5 for 16 partitions. However, the random partitioning caused many interpartition messages. When using higher communication to device evaluation cost ratios, the other heuristics performed better. Smith *et al.* found

that the concurrency fell dramatically as the communications cost increased.

Agrawal also ran computer simulations on several test circuits using random partitioning, and concluded that the level of concurrency was dependent on the level of circuit activity with values ranging from 3.7 to 7.6 for 10 partitions [Agrawal, 1986]. For large numbers of partitions the concurrency saturated because of the higher variance in circuit activity among the partitions. Agrawal described a partitioning heuristic, similar to trace partitioning, for multiple-delay simulation, where multiple fan out gates can reside in the same partition if they have different delays. The heuristic also tries to balance the number of gates in each partition.

Soulé and Blank implemented three parallel algorithms for logic simulation on a 16 processor, shared memory Encore Multimax [Soulé & Blank, 1988]. The shared memory stores the circuit description. The first algorithm was a *synchronous* event-driven algorithm (also called time wheel simulation), where processors process all events in one time step before moving onto the next time step. When using a single event queue the speedup was 2 with 8 processors. To avoid queue contention, Soulé and Blank used distributed queues, where each processor had one queue for each of the other processors. When a processor generates a new event for a node, it picks another processor in round-robin fashion, and adds the event to its queue on that processor. If a processor runs out of events on its incoming queues, it can take events from queues on other processors. Soulé and Blank achieved speedups of about 6 with 8 processors, and found that using larger numbers of processors to increase speedup required high levels of circuit activity.

Soulé and Blank implemented a compiled mode parallel simulator that evaluated every element at every time step. They statically partitioned the elements among the processors of the Encore Multimax and achieved high processor utilization rates, but most of this is unnecessary work as the out-

puts of most of the elements remain unchanged each time step. They also developed an *asynchronous* event driven algorithm to avoid the overheads and waste of idle processors, associated with synchronizing at every time step. Normally asynchronous algorithms have problems when a processor simulates too far ahead in time, and receives an event from the past. When this happens a processor must *rollback* the state of the circuit to that time, and cancel any events that it has generated since then [Jefferson, 1985]. Handling the rollbacks limits the advantage of simulating ahead of time. The method also needs more memory for state storage. Soulé and Blank's method works on a logic element by logic element basis. A processor evaluates the output of an element as far into the future as possible over the time where it knows the behaviour of all the input nodes. It then queues all the fan-out elements for evaluation on queues located on other processors. Feedback paths reduce the algorithm to a standard synchronous event-driven algorithm, which reduces the parallelism if the feedback path includes a large portion of the circuit. Soulé and Blank quoted experimental results that show a performance improvement using the asynchronous algorithm, with speedups up to 11 using 16 processors.

Bailey and Snyder discussed synchronous and asynchronous synchronization methods. and concluded that asynchronous will perform better or equal to synchronous algorithms [Bailey & Snyder, 1989].

Soulé and Gupta evaluated the suitability of the Chandy-Misra algorithm [Chandy & Misra, 1981] for parallel logic simulation, using a shared memory Encore Multimax with 16 processors [Soulé & Gupta, 1989]. In the Chandy-Misra distributed-time discrete-event algorithm, each logic element uses a local clock and sends time-stamped messages to the elements connected to it. A logic element updates its local clock and output when its receives time stamped messages on all its inputs. A logic element only sends a message when its output changes. Deadlock occurs when each element has an input

with no pending events. The algorithm can resolve this by scanning all unprocessed events in the system and finding their minimum time-stamp. It uses this to update the valid-time of all logic elements with no inputs at that time. Thus the algorithm oscillates between a compute phase and a deadlock resolution phase. Soulé and Gupta discussed the selective use of NULL messages (sent when an output remains the same after advancing the local clock) to reduce the numbers of deadlocks to improve performance. They reported that the original Chandy-Misra algorithm generated an average theoretical concurrency of 68 for industrial test circuits. When the overheads of synchronization and scheduling are included, however, only a 20-fold speedup is likely. Soulé and Gupta discussed techniques that may improve this by a factor of 3 or 4.

Subramanian and Zargham discussed the use of demand driven techniques for parallel logic simulation which they implemented on a shared memory Sequent Balance 8000 multiprocessor [Subramanian & Zargham, 1990]. Conventional discrete-event simulation evaluates the values at all nodes in the circuit even though the designer may only be interested in the signal values at a particular node at a particular time. Demand driven techniques only evaluate nodes that are necessary to evaluate the node signals requested by the designer. They can save substantial computation when the number of output pins of interest are small and the time intervals at which values are needed are small. Demand driven techniques have the disadvantage of requiring memory to store the output values of intermediate nodes for the duration of the simulation.

The algorithm used by Subramanian and Zargham uses demand driven techniques to preprocess the circuit information, in response to the required output information, to reduce the number of input events and select the logic elements that it needs to evaluate. It then carries out distributed-time discrete-event simulation (Chandy-Misra technique) using the preprocessed

information, avoiding the large memory requirements of pure demand driven simulation. Their results indicated a two to three times performance improvement over pure discrete event simulation. Using up to 5 processors, they achieved parallel speedups of about 3.

Mueller-Thuns *et al.* looked at the effect of communications speed on the speedup available from parallel logic simulation [Mueller-Thuns et al., 1989]. They defined a *communication to computation ratio* r to be the ratio of the cost of synchronizing two processors (achieved by sending a short message in a multicomputer) to the cost of evaluating a logic element. They plotted theoretical curves of speedup versus the number of processors (up to 16), for $r = 3, 5, 10$ with maximum speedups of 4.5, 6.5, and 8.5 respectively. For $r = 10$, the speedup reached a maximum with about 10 processors and then began to decline.

Chamberlain and Franklin plotted similar theoretical curves for hypercubes with up to 60 processors, with equivalent values for r of 0.75, 3, and 9 [Chamberlain & Franklin, 1988]. Their results showed a 50% speedup improvement for large numbers of processors when reducing r from 3 to 0.75. Curves with $r = 9$ reached a maximum speedup of less than half those of $r = 3$; this maximum occurred with about 40 processors.

Smith *et al.* derived theoretical speedup results for several communication networks, with equivalent values for r of 1/3.1, and 3 [Smith et al., 1988]. Their results showed speedups with 16 processors connected with a crossbar switch, using random partitioning, of 10.1, 4.7 and 1.6 respectively. With values of r above 1, they found that more complex partitioning heuristics were necessary to improve the speedup (e.g from 1.6 to 2.4 with $r = 3$). With $r = 1/3$, random partitioning gave the best results. This was most noticeable when using a crossbar switch to handle the extra interpartition messages; the crossbar switch improved speedup by about 30% over a hyper-

cube connection.

Using the above definition of the communication to computation ratio r, the HPC architecture (discussed in Section 2.2.5.1) has an r of about 3. This assumes that short messages are sent where the latency to set up the message dominates the message time. Better performance will result if the data for several messages is sent as a single message, taking advantage of the available communications bandwidth.

The theoretical simulation of the effect of communication speed show that r must be less than 3 to make effective use of more than 10 processors. Much lower values of r can simplify the task of partitioning the logic circuit among the processors and can allow for a better load balance.

3.4.3.4 Fault simulation

Multiple processors can speed up fault simulation using similar circuit partitioning techniques to logic simulation, except the amount of data transferred between partitions is much greater. Other techniques include partitioning the list of faults to simulate, or partitioning the test vectors.

Levendel *et al.* discussed a multiple, special purpose processor implementation of a parallel fault simulation algorithm (simulates multiple faults in one simulation pass, see Section 1.4.2.4) [Levendel et al., 1983]. Their strategy was to partition the circuit into subcircuits for each processor, using similar considerations as in for parallel logic simulation.

Markas *et al.* discussed using a network of workstations to speed up concurrent fault simulation (see Section 1.4.2.4) [Markas et al., 1990]. They looked at either partitioning the fault lists or partitioning the test vector list. Both methods involve minimum interprocessor communication, which suits a distributed network implementation. Markas *et al.* pointed out that

partitioning the test set can cause problems for sequential circuits where the order of applying test patterns is crucial. Test list partitioning may also result in redundant work when more than one test vector can detect a particular fault. Markas *et al.* chose fault list partitioning, and evaluated several partitioning schemes. They incorporated techniques to dynamically load balance calculations, and recover from breakdowns in the distributed computing network. They achieved near linear speedups for up to 4 workstations, but the speedup dropped off with more workstations, because of smaller fault lists per processor.

3.4.3.5 Summary

Researchers have used shared memory machines for parallel direct-method circuit simulators. The machines use multiple processors to operate on a shared matrix representing the set of circuit equations. They have also taken advantage of vector processing facilities on the individual processors. These implementations have used less than 8 processors and, although speedups are close to linear, they do not offer hope for simulating large circuits in reasonable time.

Relaxation techniques allow partitioning of the circuit into subcircuits, allowing multiple processors to simulate each subcircuit independently. These techniques suit a distributed memory implementation, where the limitations to performance are the interprocess communication speed, and the quality of the load balance. Circuits that have large groups of tightly coupled nodes are hard to partition for uniform load balance, and subcircuits with many external interconnections, place loads on the communication network.

Parallel logic simulation involves partitioning the circuit, and simulating each partition using an event driven approach that takes advantage of the low levels of activity in a circuit. This again suits a distributed memory archi-

tecture. Measurements of the maximum level of concurrency, assuming ideal communication of results between circuit partitions, show a maximum level of concurrency of about 10. The level of concurrency depended on the level of circuit activity. Using large numbers of partitions led to a saturation of concurrency because of a higher variance of circuit activity.

Parallel fault simulation can use similar circuit partitioning techniques as for logic simulation, although the communication requirements are more demanding, and may not suit a distributed memory architecture. Alternative techniques include partitioning either the fault list or the test vector set among processors, which suits a distributed implementation, and provides potential for high speedups for large problems.

3.5 Conclusions

MIMD machines are the most flexible approach to accelerating the computer-aided design process. Recent research has developed efficient parallel algorithms suitable for running on MIMD machines. In the area of computer-aided design of integrated circuits, however, the speedups obtained are generally less than a factor of 20. This is not sufficient to revolutionize the current computational limitations. For example, direct method circuit simulation is still not feasible for an integrated circuit with over 100,000 transistors, which implies that mixed-level and mixed-mode simulators will become increasingly important as circuits become larger and more complex. However, a speedup of 20 can still substantially increase the efficiency of a circuit designer. For example, tasks that normally take an hour can complete in a few minutes, allowing a more interactive design development cycle.

Special purpose accelerators now offer better performance than multiprocessors, which use standard VLSI technology, for particular applications such

as logic simulation. Large companies can afford to use these machines in addition to general purpose machines.

Advances in microprocessor design allow a MIMD machine to be built with cost effective and readily available components, to provide substantial performance improvement over a uniprocessor design workstation for a large range of CAD applications. These machines also offer performance advantages to other application areas, such as simulations of natural phenomena, allowing spreading of the development costs of communication networks, operating systems, and program development facilities.

MIMD machines with a few (less than 20) powerful processors will perform better than ones with many less powerful processors for most circuit design tasks, because of the limited concurrency available. When larger numbers of processors are present, multiple users can partition the machine into independent submachines, to make effective use of the resources. Accurate simulation also requires high speed floating point support.

Many design tasks performed on large VLSI circuits require a considerable amount of data. Many existing parallel machines only perform well when they keep all the circuit information in main memory for the duration of the algorithm, limiting their use to medium size circuits. A MIMD machine should support fast access to secondary storage to support operations on circuits with well over 100,000 transistors.

The programming model of a MIMD machine can be different from the underlying architecture, although a shared memory multiprocessor can provide a message passing programming environment more easily than a distributed memory multicomputer can provide a shared memory programming environment. Distributed memory multicomputers are simpler to build, and allow more processors for a given cost. Many parallel CAD algorithms use a distributed memory model that encourages localized computations, with a small

amount of interprocess communication, implying the use of a distributed memory machine. Some applications, however, such as simulated annealing and circuit simulation, favour large shared data structures that would be hard to store on one server processor. For these applications, a distributed memory machine should support a shared memory programming paradigm that allows the manipulation of large distributed data structures. A programming environment, supporting both shared memory and distributed memory programming techniques, will enhance the portability of parallel algorithms. Providing a standard user interface will also allow designers to use different machines without being aware of the underlying hardware structure.

Tasks such as logic simulation, which use a significant amount of interprocess communication, are highly dependent on the speed of the interprocessor communications. This implies that a distributed memory machine should use an efficient hardware supported communications structure. The use of a fast communications structure also allows the programmer to ignore the interconnection topology, greatly simplifying the programming task, and improving portability.

Many integrated circuit design teams have access to a distributed computing environment consisting of multiple workstations connected via a local area network such as Ethernet. These workstations could be part of the communications network of a MIMD machine to make efficient use of existing facilities, and take advantage of mature user interfaces. Design tasks requiring interactive input, such as schematic entry, still suit the capabilities of a uniprocessor workstation.

Chapter 4

Computer architecture for VLSI CAD

4.1 Introduction

This section will describe aspects of a computer system proposed for the computer-aided design of large integrated circuits. The system architecture consists of a communications network interconnecting high performance computer nodes and workstations to form an MIMD parallel computer. Figure 4.1 illustrates this architecture.

The high performance computing nodes will cooperate in parallel to speed up many of the compute intensive design tasks such as placement, routing, design rule checking, test generation, and simulation. Each node will have its own memory, and can operate as an independent computer. The nodes will have binary compatibility: the same machine code will run on each of the nodes, although some of the nodes may run at different clock speeds. Software support will allow programmers to use both shared memory and message passing programming techniques.

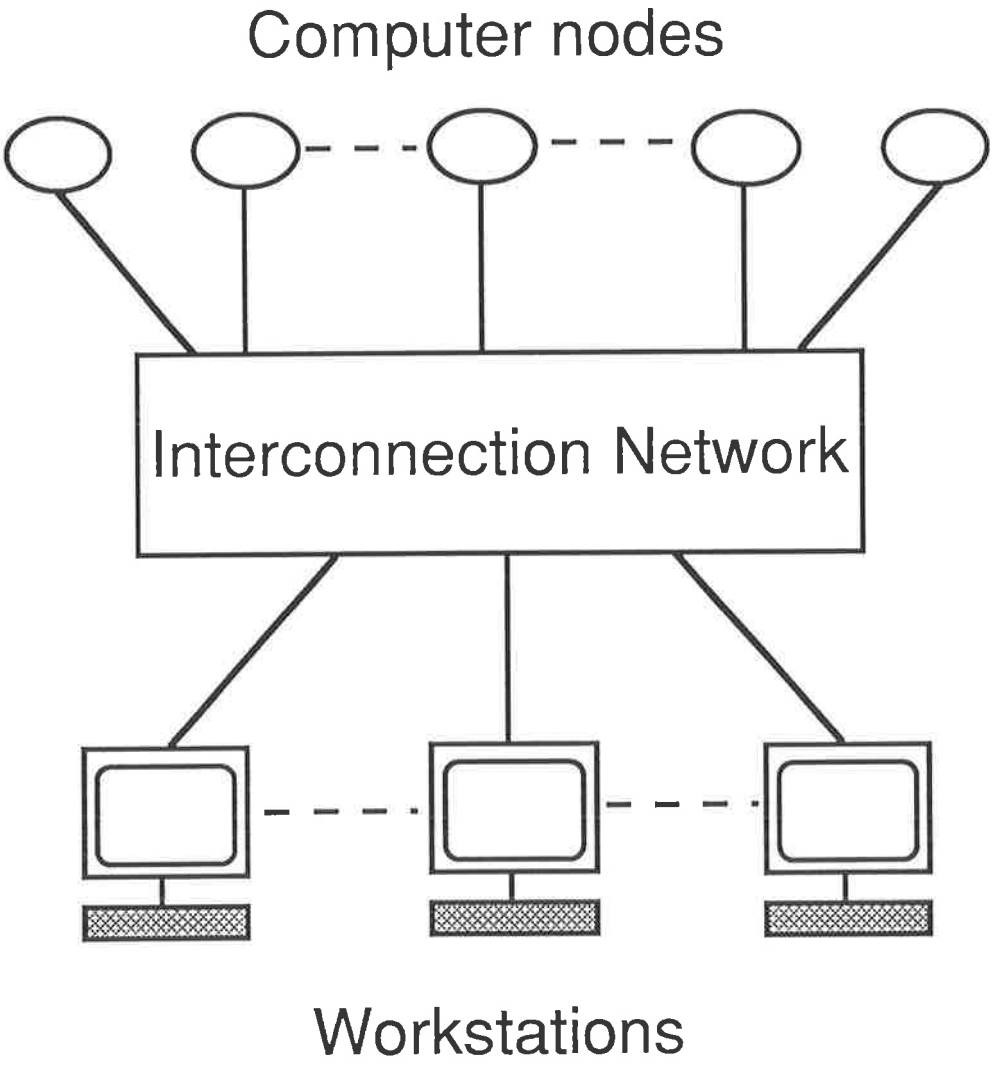


Figure 4.1: System Architecture

Special purpose accelerators, such as available for logic simulation, could connect into the network if high production volumes lead to low prices.

The workstations will provide the support for the user interactive stages of design such as schematic entry. Initially these workstations will be standard uniprocessor workstations, allowing an organization to take advantage of an existing base of workstations. As multiprocessing becomes more cost effective with high production volumes, these workstations will contain multiple processors. This will increase the demand for efficient parallel algorithms for the interactive stages of design to take advantage of the available computing resources.

Designers can have exclusive access to a workstation while sharing the pool of high performance computing nodes. Although many design tasks cannot cost effectively use more than 20 processors, greater numbers of processors can support multiple design tasks simultaneously.

The following sections will discuss the system architecture in more detail:

- workstations,
- computer node architecture,
- interconnection network,
- secondary storage,
- operating system,
- and parallel programming facilities.

4.2 Workstations

The workstations will support the wide range of existing design tools that use a graphical interface to improve the productivity of designers. Designers will use the workstations to enter circuit design details using such abstractions as block diagrams and logic gates. The workstations will provide visualization facilities for the display of diagnostic and simulation data generated by the pool of processors, and for the display of the design at different levels of the representation hierarchy. Color graphics are necessary for distinguishing features of the design.

Workstations that meet the user interactive needs of most designers are already available as high production volume items. The cost effectiveness of these machines has allowed designers to use them as single user machines. Exclusive use of a workstation allows a designer to get a fast response to commands, hence improving the productivity of the designer.

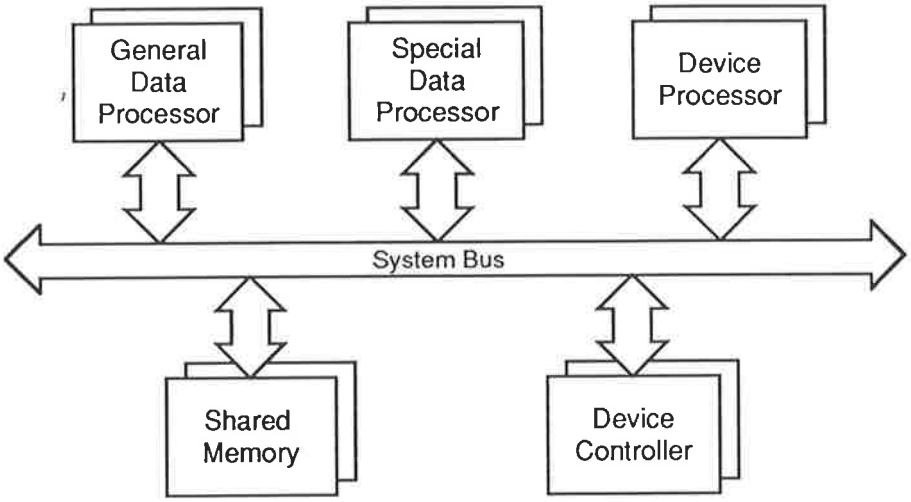
Several researchers have examined the effect of the response time to commands on the productivity of a user. Thadhani collected results that show that as the system response time for user interactive tasks decreases below 1 second, the productivity begins to rise sharply [Thadhani, 1981]. A user's productivity with a 0.3 second response time was about double that with a 3 second response time. Further work at IBM [Brady, 1986][Hennessy & Patterson, 1990, pp. 509-510] has shown that an expert user can improve productivity by a factor of 8 at 0.3 seconds response time, compared to the productivity at 1.5 seconds response time. A novice user can equal the productivity of an expert when the novice has access to a workstation with faster response time. Brady has developed a model that attempts to explain the effect of response time on a user's behaviour [Brady, 1986]. Brady also notes that as the response time decreases, the efficiency at which a user can enter a command becomes more important.

As the cost of workstation hardware has diminished, the time a designer spends on a design becomes more dominant in the overall cost of the design. Thus the improvement in productivity available with a fast response time justifies the exclusive use of a workstation by a designer, even though the workstation may lie idle while a designer thinks about the next step in the design process.

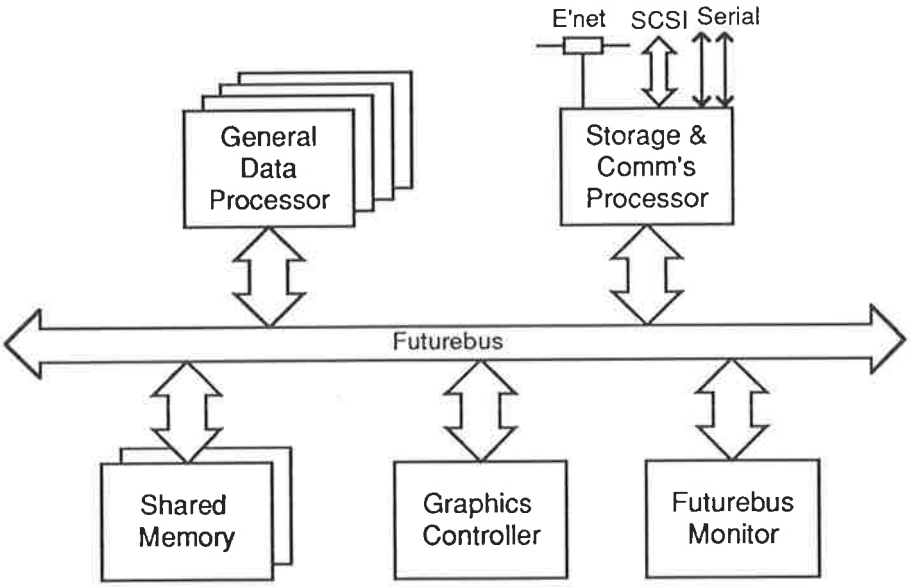
The workstations will also provide the facilities necessary to develop software to run on the pool of processors. Computer-aided design techniques, such as automatic synthesis, which have greatly improved the productivity of circuit design, are now improving the productivity of software development [Murphy & Voelcker, 1990]. These Computer Aided Software Engineering (CASE) tools also make use of a graphical interface to help a software developer manage the complexity of software.

Architectural advances in microprocessors and their low cost (relative to multiple chip central processing units used in super-minicomputers) has led to research into the use of tightly coupled shared memory multiprocessing to improve the performance of workstations [Thacker et al., 1988; Hill et al., 1986]. These multiprocessor workstations consist of multiple processors accessing shared memory via a shared bus. The processors each have a cache to minimize the traffic across the bus. The machines use cache coherency techniques to ensure that all processors have a consistent view of memory [Stenström, 1990].

The Leopard workstation, implemented at Adelaide University, is an example of such a multiprocessor workstation [Ashenden et al., 1987]. Figure 4.2 illustrates the architecture of the Leopard workstation. Up to four General Data Processors do general computation tasks. The operating system can allocate a task to any of these processors, and the operating system kernel is distributed across the processors [Vaughan et al., 1988]. The processors



Leopard Workstation Architecture



Leopard - 2 Configuration

Figure 4.2: Leopard multiprocessor workstation [Ashenden et al., 1989]

consist of National Semiconductor NS32532 microprocessors with 512 Kbytes of cache memory, and 4 Mbytes of local memory for local operating system code and data. The processors access 16 Mbytes of shared memory over a bus designed to the IEEE Futurebus standard. They use caches to reduce bus traffic,,with bus snooping hardware to maintain coherency [Ashenden & Marlin, 1988]. The Special Data Processors can be additional special purpose processors for accelerating a particular task, such as image processing or logic simulation. The Device Processors are processors for handling particular external devices such as disk drives, network interfaces, and graphics terminals. The Device controllers are non-intelligent device interfaces controlled by tasks running on the General Data Processors.

Multiprocessor workstations are becoming commercially available (such as the SPARC[1] compatible workstations from Solbourne Computer, Longmont, Colorado). Most of the software on these machines does not take advantage of the multiple processors to reduce the running time of a single application. The multiple processors mostly provide a faster response time for a group of independent applications running at once. As higher numbers of these workstations reach the marketplace, there will be incentive for software developers to begin producing software that uses parallel processing techniques to reduce the running time of single applications. This will also provide incentive for developing computer-aided software engineering techniques to ease the task of writing parallel software.

The UNIX™ operating system is becoming the dominant operating system for workstations used for computer aided design. There have been recent moves to try to standardize the operating system [Murphy & Voelcker, 1990]. At the same time there has been much emphasis on providing a standard graphical user interface to prevent users having to learn a new interface for

[1]SPARC is a trademark of Sun Microsystems, Inc.

each new software application [Harrison et al., 1990].

As multiprocessor workstations come onto the marketplace, efforts are also underway to provide an efficient multiprocessor version of the UNIX™ operating system. Early attempts at this have simply run the operating system kernel on one processor, which becomes a bottleneck when the other processors need to make frequent calls to operating system functions. More recently the operating system kernel is distributed among the processors [Cajani, 1987; Russell & Waterman, 1987].

Thus workstations suitable for supporting the user interactive stages of integrated circuit design will need to support industry standard operating systems and graphical user interfaces [Harrison et al., 1990]. This allows designers to take advantage of the large base of software developed to run on such workstations. The performance of these workstations must be sufficient to allow designers to receive responses to their commands before they can become distracted from their design task. To make the graphical interface productive, the workstations must be able to update the graphics display almost instantaneously in response to simple graphical interface commands (such as moving windows). Otherwise the time in waiting for window updates will become a distraction to the designer. and remove many of the advantages of a graphical interface.

As the costs of multiprocessor workstations decrease, and the software technology for making use of multiple processors improves, multiprocessor workstations will become the workstations of choice.

4.3 Computer Node Architecture

As discussed in Chapter 3, most current parallel algorithms for the design of integrated circuits have only achieved small scale speedups over uniprocessor algorithms, generally less than a factor of 20. Thus the system will perform most efficiently with low numbers of high performance processing nodes, rather than with hundreds of simple processing nodes. It is economical to spend more on the performance of these nodes, because designers in a design team share the high performance nodes.

The requirements of the computing nodes suitable for CAD of integrated circuits are:

- VLSI microprocessor technology.
- high speed floating point support,
- memory management to provide virtual address spaces,
- and large amounts of local memory.

The following subsections will discuss these requirements in more detail.

4.3.1 VLSI Microprocessor Technology

To remain cost effective the nodes should take advantage of the developments in VLSI microprocessor design, rather than use expensive VHSIC technology (see Section 3.3.1). These developments have allowed microprocessor manufacturers to incorporate many of the functions and techniques used in mainframes and supercomputers onto a single chip at much lower cost.

Typical functions found on a single chip microprocessor include data and instruction caches, and memory management. Recent RISC based micro-

processors have included multiple pipelined functional units, and can issue multiple operations per cycle (see Sections 3.2.2.2 and 3.2.2.3) [Jouppi et al., 1989]. These microprocessors make use of instruction level parallelism, independently of the programmer, with the aid of compilers. They achieve high performance on program code that is amenable to vector processing techniques for little extra cost, while still providing good performance for scalar code.

The Intel i860 microprocessor is a recent example of these microprocessors [Kohn & Margulis, 1989; Piepho & Wu, 1989; Margulis, 1990]. Figure 4.3 shows the layout of the 1,000,000 transistor, 10mm × 15mm chip. It has a RISC based integer core, a 3-stage pipelined floating point multiplier, a 3-stage pipelined floating point adder, an instruction cache, a data cache, a memory management unit that supports paged virtual memory, and a 64-bit external data bus. Using the 64-bit wide instruction cache it can execute one 32-bit integer instruction and one 32-bit floating point instruction per clock cycle. The floating point adder and multiplier can operate in parallel. A compiler can make use of the pipelining to get results from floating point adds and multiplies at the rate of one per clock cycle once the pipeline is full. The 64-bit wide external data bus can read in a 64-bit word every two clock cycles.

The performance of the i860 integer unit is about 30 MIPS when running with a clock rate of 40 MHz (where one MIP is equivalent to the performance of a VAX 11/780, actual numbers will vary depending on the code being run). The performance of the i860 on floating point code is highly dependent on the quality of the compiler. Margulis claims performance above 10 MFLOPS for the LINPACK benchmark [Margulis, 1989]. With a non-vectorizing C compiler the performance may be half that. To get the best performance, a library of hand optimized routines may be necessary to exploit the internal parallelism of the microprocessor.

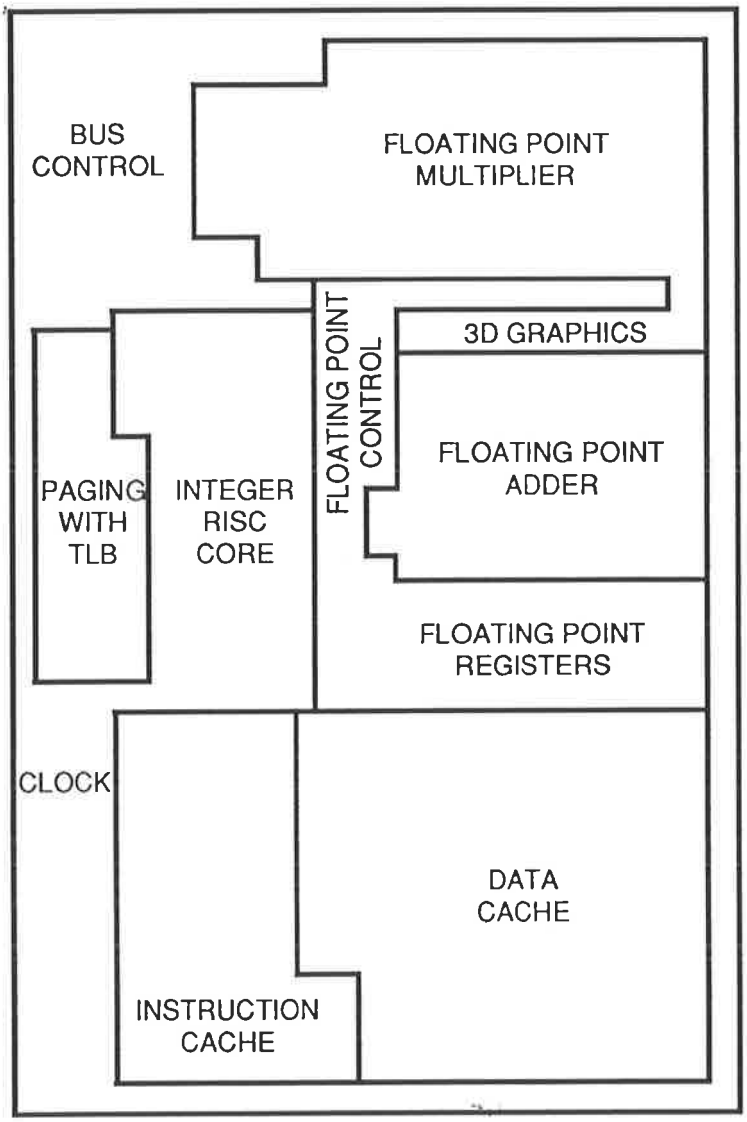


Figure 4.3: Intel i860 microprocessor layout [Kohn & Margulis, 1989]

Thus recent microprocessors, running at higher clock speeds, run integer code about 8 times faster than the Motorola 68020 microprocessors operating at 16.6MHz on the HPC (see section 2.2.5.1). Floating point performance has increased by about 30 to 60 times that achievable with the 68881 coprocessor used with the 68020 microprocessor. Most of this performance improvement is a result of integrating the floating point unit onto the microprocessor.

The most important parameter in choosing a microprocessor for the high performance nodes is the performance on scalar code. However, the potential for higher performance on vectorizable code is advantageous for applications such as direct method circuit simulation if the additional cost is minimal.

4.3.2 Floating Point Support

High speed floating point support is necessary for accurate circuit simulation. Previously high performance microcomputers used separate chips for this, but increasing levels of integration have made it possible to incorporate this on the microprocessor chip. Incorporating the floating point hardware on chip allows for larger bandwidth data paths between cache memory and floating point registers. Multiported register sets allow the integer processing unit to update the contents of some floating point registers, while floating point operations are taking place.

4.3.3 Virtual Memory Management

Memory management facilities allow programs to use large virtual address spaces without knowledge of their physical location in memory. An operating system can then decide where to load them in memory and can relocate them as necessary. Memory management hardware can save on the overhead of translating from virtual to physical addresses by using a cache of recently

used mappings, called a translation lookaside buffer (TLB) [Denning, 1970]. This hardware is now commonly available within a microprocessor. Providing these facilities on a high performance node gives the operating system flexibility in managing the physical memory available, and allows for the use of a disk to provide demand paged virtual memory.

Milenkovic provides a good overview of virtual memory management techniques, and reviews several recent examples of commercial memory management hardware [Milenkovic, 1990].

4.3.4 Local Memory

Each processing node should have a large amount of local memory to support large programs, and allow large data structures. This means that high density dynamic RAM is necessary. Although recent microprocessors have internal caches to prevent external memory accesses becoming a bottleneck, another level of caching between the microprocessor and the dynamic RAM can be beneficial. Dynamic RAM chips (such as those supporting page mode and static column accesses) can achieve bandwidths matching recent microprocessor bus bandwidths for multiple accesses to single rows in their memory arrays. Dynamic RAMs still suffer higher latency to access the first bit compared to a static RAM. High speed static RAM can act as a zero latency cache for the microprocessor to minimize the effects of the dynamic RAM latencies.

The goal is to provide as much local memory as possible given area and cost constraints [Kung, 1986]. Keeping local memory on the same board as the microprocessor avoids the overheads of accessing memory over a bus. Commercially available single board computer modules (double-eurocard size), such as available for VMEbus, typically have 4 Mbytes of on-board local memory using 1 Mbit DRAMs. 4Mbit DRAMs offer the potential for up to

16 Mbytes of on-board local memory. Larger board sizes used with standards such as IEEE Futurebus can support additional memory.

The steady improvement in storage densities (DRAM density has quadrupled about every 3 years, and magnetic disk density has doubled about every 2 years) shows that additional address bits will be necessary on the next generation of microprocessors to allow the possibility of address spaces above 4 Gigabytes. Hennessy and Patterson report that the average memory required by a program has increased by a factor of 1.5 to 2 per year [Hennessy & Patterson, 1990, pp. 16-17].

4.4 Interconnection Network

This section will start by discussing a network design for heterogeneous multicomputers called Nectar. The Nectar design goals closely match the design requirements of the network required to interconnect the computing nodes in Figure 4.1. In addition, the Nectar design has addressed the main sources of inefficiency that cause long message latencies.

This section then goes on to discuss the trend in network topology back to networks with higher diameter, and dimensions that suit physical construction in 2 or 3 dimensions. Next, it discusses an emerging interconnection standard called the Scalable Coherent Interconnect (SCI), for connecting processors in a large high performance multiprocessor system. It will also introduce two more emerging standards for interconnecting computers: the HSC standard and the FDDI standard. Finally it will discuss the requirements of the interconnection network for interconnecting the workstations and computing nodes; the discussion takes into account the recent advances discussed in the preceding sections.

This section consists of the following subsections:

- Nectar,
- Network topology,
- Scalable Coherent Interconnect (SCI),
- High-Speed Channel (HSC) standard,
- Fiber Distributed Data Interface (FDDI) standard,
- and Interconnection network requirements.

4.4.1 Nectar

Arnould *et al.* discussed the design of a network for heterogeneous multicomputers called Nectar [Arnould et al., 1989]. The three major goals for their interconnect network were

- to support heterogeneity,
- to provide scalability,
- and to provide low latency, high bandwidth communication.

Different parts of an application may have different computing requirements, therefore a network should support different types of nodes, hence making it heterogeneous. For example, in computer aided design the computing requirements of design specification are different from those of design simulation. A conventional workstation can support the entry of design details, and a high performance computing node can support simulation.

A scalable network allows the addition of nodes without major alterations to the computing system. The addition should not affect the communication latency and bandwidth of existing nodes, and should not require major alterations to the system software. Most networks are only scalable within a

limited range. For example a network designed to handle about 100 nodes may not scale to 100,000 nodes, because of the effects of the extra distances signals must travel.

The latency to set up the communication path and handle the communication protocols usually dominates the time to send a short message between nodes. The bandwidth of the communication link dominates the time to send a long message. Both low latency and high bandwidth are necessary to support communications for a variety of applications.

Annaratone *et al.* have simulated the effect of different communication speeds on the performance of a variety of applications [Annaratone et al., 1989]. They defined a communication ratio q as the ratio of the time to send or receive a double precision floating point number over the time to do a double-precision multiply-add operation. They examined the effect of varying this ratio for common applications using 64 processors connected together in a torus arrangement. They concluded that a $q < 8$ was necessary for high performance. The performance degrades quickly for ratios above 8, whereas ratios below 8 only achieve marginally improved performance.

On the HPC architecture discussed in Section 2.2.5.1, q is about 60 using the above definition. The latency to set up the message dominates this ratio, rather than the network transmission time. If a processor node can do several floating point operations and send the results as a single message, q can drop to about 1.3. Thus an application should try to send data in a few large messages, rather than many small messages, to make use of the communication bandwidth available.

A communications system can increase its bandwidth more easily than it can reduce its latency. A system can increase the width of its data path, or use high bandwidth media like fibre optics, to increase the bandwidth. Communication software tends to dominate the latency, and this is hard to

speed up.

For example, the transfer rate between two processes running on separate processors (25MHz 68020) of the HPC, using the VORX communication primitives, is about 1 Mbytes/sec over a communication link capable of 12.5 Mbytés/sec [Gaglianello et al., 1989]. The time to transfer 4 bytes is about 300 microseconds; most of this time is for the software overheads.

Arnould *et al.* state the three main sources of network inefficiency as

- context switching and data copying,
- overhead of high level protocols,
- and interrupt handling and header processing for each packet.

Context switching overhead occurs when a user process invokes an operating system function to do a network transfer. Often the operating system function copies data from user space to the kernel space before doing a transfer. A communications function uses high level protocols to ensure reliable communications; these add to latency. Normally the node processor executes the high level protocol software. The reliable transmission of a message may require handling several control packets, resulting in interrupting the node processor to do the necessary processing.

Nectar provides a communications accelerator board (CAB) to handle the network tasks for each network node. User processes have direct access to a high-level network interface mapped into their address spaces, removing the need for system calls. The CAB handles the communication protocols, and informs the node processor when a complete transaction is successful. The disadvantage is the added cost of the CAB to each processing node, as the CAB contains a SPARC™ processor along with memory and a DMA controller.

A recent paper by Ramachandran *et al.* also suggests using hardware acceleration for the communications [Ramachandran et al., 1990]. They suggest that hardware should accelerate local communications as well as network communications. This is because recent message-based operating systems benefit from faster local communications between client processes and server processes.

The Nectar network consists of HUB crossbar networks with fibre optic lines (with 100 Mbits/sec peak bandwidth) to the CAB boards. Figure 4.4 illustrates a single HUB cluster. Initially each HUB consisted of an 8-bit wide 16×16 crossbar switch. Arnould *et al.* report that 8-bit wide 32×32 crossbars are practical using off-the-shelf components, and 128×128 crossbars are possible using custom VLSI. The HUB clusters can interconnect in any topology to suit the application. Communication between nodes on separate HUBs can use either circuit or packet switching (see Section 3.3.6) to route data through the HUBs.

The HPC network (see Section 2.2.5.1) uses similar ideas with clusters of 12×12 crossbar connected nodes interconnected in a hypercube topology [Gaglianello et al., 1989]. The network uses hardware routing using virtual cut-through (Section 3.3.6), but relies on the node processor for processing the communication protocols.

As the processing speed of microprocessors has increased (see Section 4.3.1), the speed of communications must also increase to prevent communications becoming the bottleneck. For example, processing speeds for integer code have improved by about a factor of 8 over the processors used in the HPC. As the microprocessor on an HPC node handles the communications protocols, the software latencies will also scale with advances in processing speed (if memory speed has also increased). The bandwidth of the communications links will need to increase to match increases in processing speed, or it will

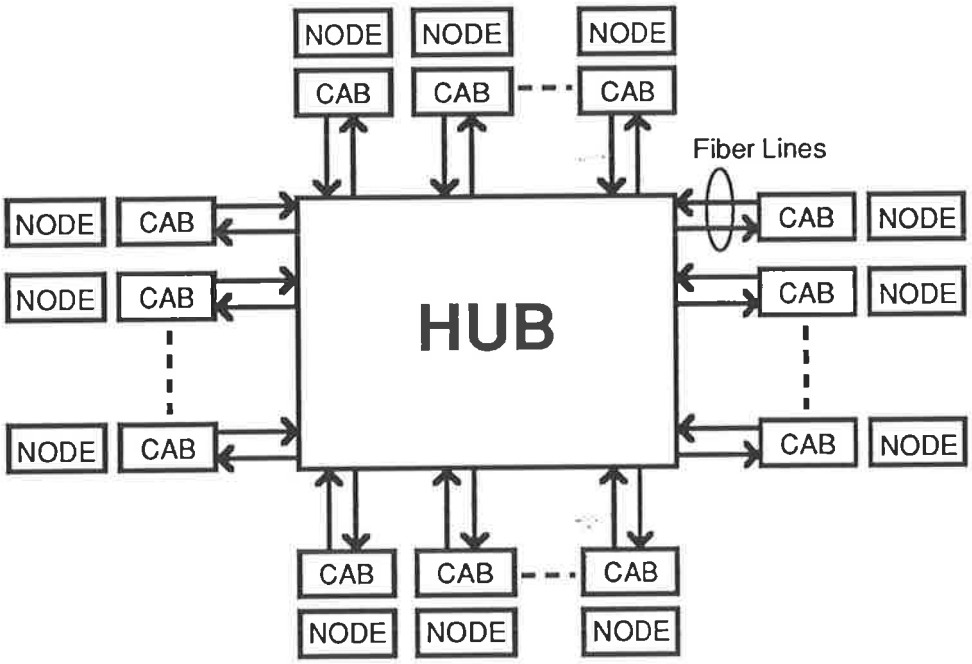


Figure 4.4: Single HUB Cluster of Nectar network [Arnould et al., 1989]

become the bottleneck instead of the communications software.

The following subsection discusses trends in the network topologies used to build multiple processor machines.

4.4.2 Network Topology

Most commercial multicomputers such as the Intel iPSC and the NCUBE have used a hypercube interconnection topology. The topology has the advantage of reducing the number of intermediate nodes that a message must traverse in the worst case, compared to networks such as meshes and trees. Programs written for these machines must often take into account the network topology to maximize the performance.

Many real world problems, including static analysis problems (see Section 3.4.2), suit a grid like mapping onto a parallel machine. Annaratone *et al.* pointed out that this mapping onto a hardware hypercube topology is only efficient if the communication time is low relative to the calculation time (*i.e.* ratio of communication time to calculation time less than 8) [Annaratone et al., 1989].

Ideally parallel programs should run on any MIMD machine, irrespective of the underlying network topology. Athas and Seitz pointed out that advances in network design, such as the use of wormhole routing techniques, are making networks fast enough for the programmer to ignore the underlying architecture [Athas & Seitz, 1988]. Efficient routing techniques have removed some of the problems of traversing many intermediate nodes. Recently researchers have favoured network topologies such as rings and toruses, which have higher diameter, and dimensions that suit physical construction in two or three dimensions (higher radix versions of k-ary n-cubes) [Annaratone et al., 1989: Vitányi, 1988]. Athas and Seitz reported a result derived by Dally

[Dally, 1987] stating that a two dimensional network minimizes network latency for 256 processors.

Two dimensional networks such as meshes are easier to wire together, because individual wires are short and do not need to pass around or over other processors.' This allows more parallel wires for the same path, providing higher bandwidth, in contrast to hypercube topologies that need longer wires. Breaking toruses into 2-D meshes makes it easier to route messages and partition the network into equivalent subnetworks for multiple users. Athas and Seitz predict that 2-D networks will be standard for the next generation of multicomputers [Athas & Seitz, 1988].

The following subsection discusses an emerging interconnection standard, which provides for high bandwidth point to point links to allow systems to use simple topologies such as rings.

4.4.3 Scalable Coherent Interconnect (SCI)

Two standards are emerging for high performance multiprocessor systems with shared memory:

- Futurebus,
- and Scalable Coherent Interface.

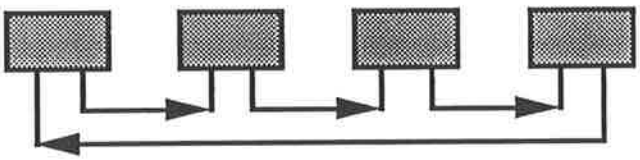
Futurebus+ is a high performance bus, which supports cache coherence, for small scale multiprocessor systems [IEEE P896.1, 1990]. Using asynchronous handshaking, Futurebus+ is expected to support burst transfers of between 20 and 25 million transfers/sec. In source-synchronous mode, Futurebus+ should support greater than 50 million transfers/sec, where the standard supports bus widths of 32, 64. 128, and 256 bits [Hawley, 1989].

The SCI (Scalable Coherent Interface) standard defines a transaction set and protocols that support a variety of communication technologies [IEEE P1596, 1990; Warren, 1990]. The aim of SCI is to provide a coherent shared memory model scalable to 64,000 nodes.

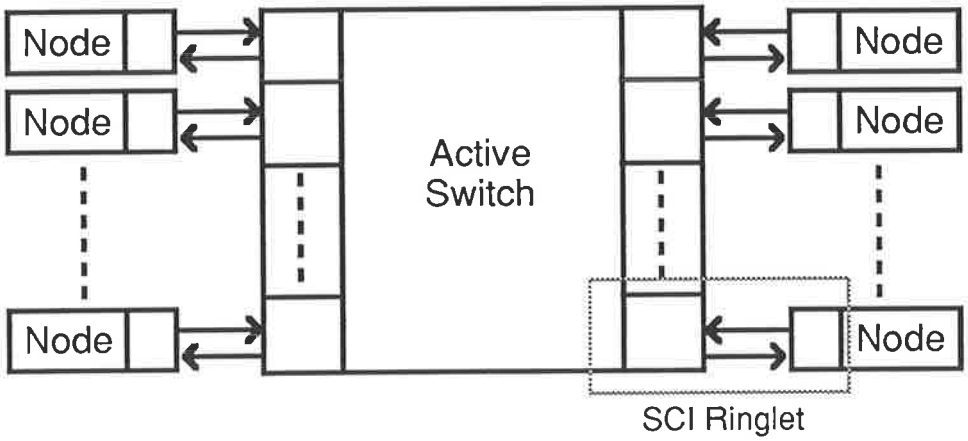
The SCI uses unidirectional, point-to-point links instead of a backplane bus such as Futurebus. Modules connecting to a shared bus disturb the electrical transmission characteristics of the bus. limiting the bandwidth of the bus. Also a bus only supports one transaction at a time, which can cause a bottleneck as the number of processors increases. Point-to-point links allow a higher clock rate, hence saving on the number of signals in the link for the same effective bandwidth. Reducing the number of signals in a link, reduces the pin count for bus interface logic. This allows the integration of the entire bus interface on one chip, effectively improving the matching between signal lines, thus reducing the skew and allowing higher clock rates. An experimental implementation of the SCI standard uses 16-bit wide point-to-point links on an ECL/copper-backplane to achieve a link speed of 1 Gigabyte/sec, using both edges of a 250 MHz clock.

A low cost SCI system would use a passive backplane with the processors connected in a ring topology using unidirectional links. A large scale, high performance system would use an active, multistage switching network with SCI interfaces at the node and at the node's port into the switching network. Each interface includes a response and a request buffer. Figure 4.5 illustrates these two possible applications of the SCI standard.

The SCI protocols are packet based and support shared memory transactions. The protocol uses 64-bit wide addresses, with 16 bits used as node identifiers and the remaining 48 bits available for addressing within a node. Transactions include reads and writes. A *requester* begins a transaction and a *responder* completes it. A read transaction begins with the requester send-



Backplane Ring



SCI ringlets connected with Active switch

Figure 4.5: SCI applications

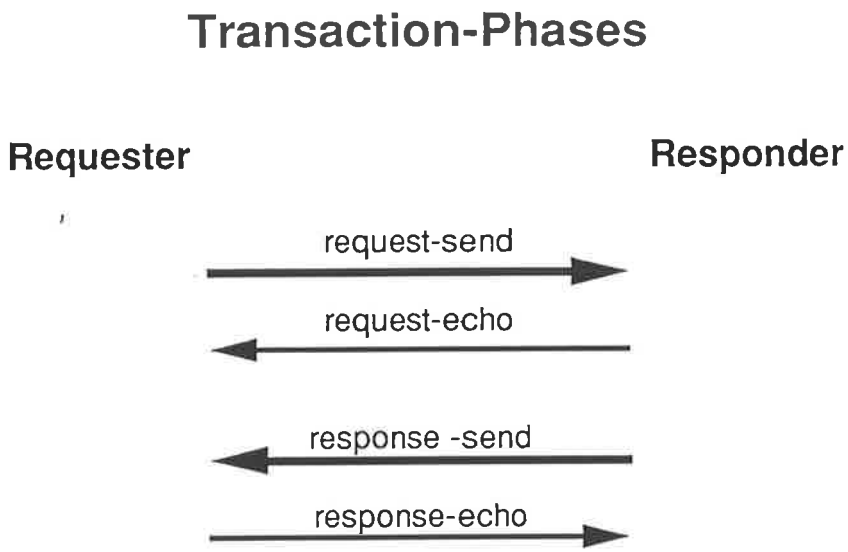


Figure 4.6: SCI Transaction phases [IEEE P1596, 1990]

ing a request packet into the network. Intermediate nodes or switches route the packet by examining the destination address. If the destination node has space for the packet, it will send a done-echo packet: otherwise it will send a retry-echo packet to the requester, informing the requester that the request packet must be resent. When the destination is ready, it will send a response packet containing the requested data. The requester can either confirm receipt of the data, or request that the responder resend the data packet. Thus the protocol uses a minimum of four packets for a read transaction. Write transactions occur in a similar manner. Figure 4.6 illustrates this protocol.

The SCI maintains cache coherence in a shared memory environment by using a directory-based cache coherence protocol [Agarwal et al., 1988; Chaiken et al., 1990; James et al., 1990]. Every memory block (usually the size of a cache line) has a directory, with a pointer to a linked list of processors sharing it. When a processor wants to modify a cached block, it inserts itself at the head of the list and purges the remaining list entries. Thus only one processor can have write access at a time.

Protocols proposed for SCI allow for addressing processing nodes and their memory. A message passing environment is simple to implement with these protocols. There is the possibility of partitioning the memory space on each node as local and shared memory. The cache coherency protocols will allow for sharing of the distributed memory.

In a recent article, Chlamtac and Franta review recent work in high speed networks [Chlamtac & Franta, 1990]. They discuss two emerging standards for high speed networks: HSC (High Speed Channel) and FDDI (Fiber Distributed Data Interface). The next two subsections will overview these standards.

4.4.4 High Speed Channel (HSC) standard

The HSC channel standard derives from the proprietary HSX channel used in Cray computers. It uses twisted pair cable to transmit data at 800 Mbits/sec in a single direction over distances up to 25 metres. The use of fibre optics could improve on this distance. It is suitable for a circuit switching environment with networks made up of crossbar switches, and transmits data in bursts of 256 $\times$ 32-bit words followed by an error check, once it establishes a connection. Figure 4.7 shows the signals used in an 800 Mbit/sec interface.

Chlamtac and Franta report that HSC interfaces are beginning to appear on supercomputers and superminis [Chlamtac & Franta, 1990].

The next subsection will discuss a lower performance standard that allows connection over much longer distances.

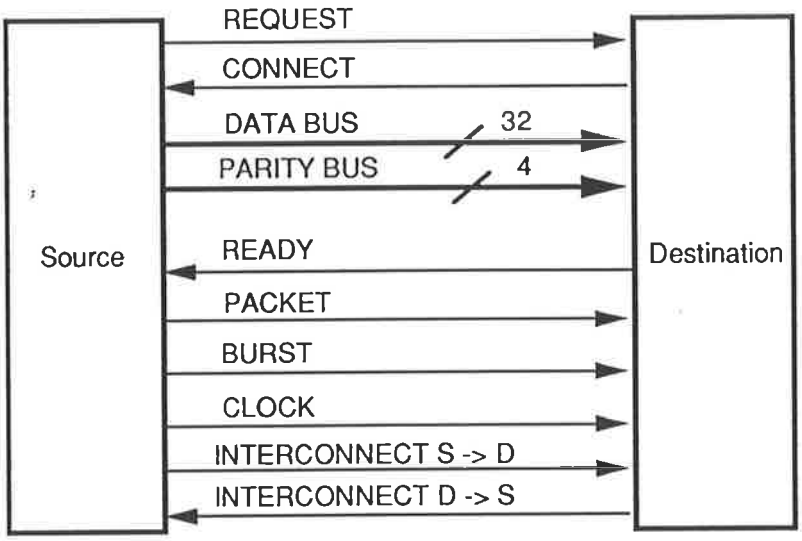


Figure 4.7: HSC Interface signals [Chlamtac & Franta, 1990]

4.4.5 Fiber Distributed Data Interface (FDDI)

Currently the 10 Mbits/sec Ethernet is the most common local area network for interconnecting workstations and their file servers [Metcalfe & Boggs, 1976]. As the performance of workstations improve, this network is becoming a bottleneck. The 100 Mbits/sec FDDI network standard [Ross, 1986] may replace Ethernet in the workstation environment. It uses token ring techniques with a fibre optic medium, and can operate over distances up to 2 km. Chlamtac and Franta report on the use of two counter-rotating token rings to provide fault tolerance [Chlamtac & Franta, 1990]. They also point out that. at present, FDDI interfaces for workstations are expensive, but should come down as their use becomes more widespread.

4.4.6 Interconnection Network Requirements

The goals for the interconnection network suitable for CAD of circuits are similar to the design goals of the Nectar network. They are as follows:

- allow heterogeneity.
- provide scalability,
- adhere to applicable standards - e.g SCI, HSC, FDDI,
- use a crossbar switch,
- and allow long distance links.

Although it is possible to build networks using ECL technology, which operate with very high clock frequencies, the cost is prohibitive. Systems with very high communication rates can use simple connection networks such as the ring without concern for the number of intermediate nodes a message must pass through. Using more complex interconnections, networks such as in the Nectar and the HPC networks (see Section 2.2.5.1) can achieve high throughput using slower communication link rates. For example a system with several hundred nodes may use a mesh connection network.

The design decision to use low numbers of high performance nodes allows the use of a crossbar interconnect for connecting the nodes. Larger scale systems can consist of a network of clusters, where each cluster consists of crossbar connected nodes. For low numbers of these clusters, a completely connected topology could also be cost effective. This will allow the network to scale up to about 100 nodes. A crossbar connected system reduces the chances of network contention for a heavily loaded system. The ability to multicast messages would also be useful. Figure 4.8 illustrates this architecture.

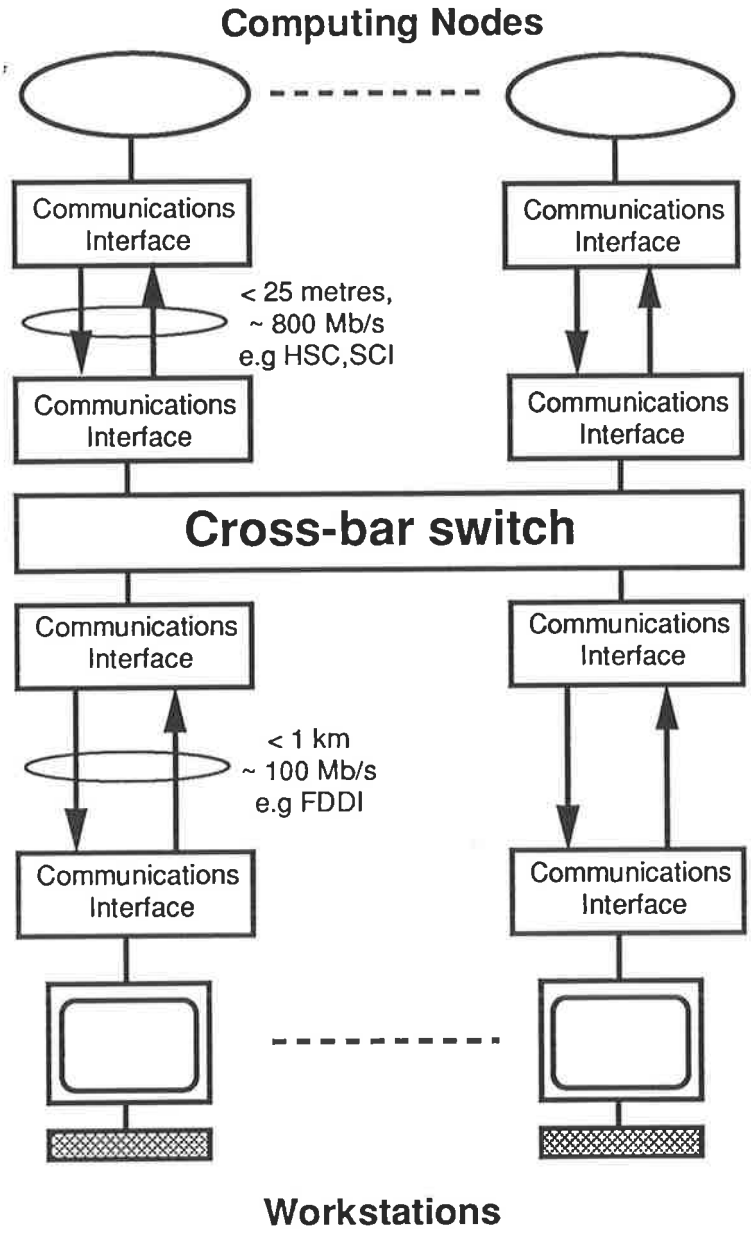


Figure 4.8: Interconnection system

A system that uses connection standards ensures compatibility with high volume production components such as interfaces for workstations. Many of the features of the proposed SCI standard are compatible with the goals above. The network can use point to point links, compatible with the SCI standard, between the crossbar interconnect and the nodes. This requires a SCI interface at both ends of the link, with two unidirectional links to make up a bidirectional communication link. The computing nodes may connect over a backplane to the crossbar interconnect to achieve high speed communication, although this makes packaging the system more difficult. Alternatively if HSC interfaces become widely available, the network could use HSC links operating at up to 800 Mbits/sec. Twisted pair cable would be adequate as the computing nodes will not need to be apart by more than 25 metres. For connecting distributed workstations, the links may be serial fibre optic links, as used in the Nectar network. These links may make use of standard 100 Mbits/sec FDDI interfaces if they become common on workstations. The system can use special purpose accelerators connected into the network with the short high speed links.

The network should provide distributed hardware routing. Programmable routing tables to route packets through the network, such as used in the HPC multicomputer, allow flexibility in adding or removing nodes from the network.

As levels of integration get higher it will become more cost effective to provide a programmable communications chip for handling the protocols. At present a high performance processing node with efficient interrupt handling can provide processing of the communication protocols. The system should give user programs access to the network interface to allow programmers to develop communications primitives specific to an application if required.

With an 800 Mbits/sec communication link, the communications software will still dominate the transfer time across the network for small transfers. The system will only achieve the communication to computation ratios required for logic simulation (see Section 3.4.3.3) and many other applications [Annaratone et al., 1989] when sending data with large transfers (i.e above 1 kilobyte). Large transfers take advantage of the available bandwidth.

4.5 Secondary Storage

A parallel application needs to receive data and instructions from secondary storage, and then return the results to secondary storage. For CAD of circuits with over 100,000 transistors, this data is large, and the time to transfer this data can form a significant proportion of the total computing time. When the size of the problem is too large for all its data to remain in the primary memory of the machine during processing, secondary storage must hold intermediate results as the calculation progresses (see Section 2.2 on Parallel Goalie). If the I/O system performance remains constant as the processing speed of the system increases, the I/O system will limit the net performance improvement.

For a distributed workstation environment, the trend has been towards using a high performance centralized file server for a network of diskless workstations. Smith and Maguire have presented results showing that accessing files on a high performance file server over a local area network is faster than accessing files on a standard local disk [Smith & Maguire, 1989].

There are several techniques, such as the use of disk caches and dynamic scheduling of accesses, available to increase the performance of a file server [Smith, 1981; Kim, 1986; Kim, 1985; Katz et al., 1989a]. The mechanical characteristics of a disk, such as the seek time, rotational latency, and transfer

rate, provide an upper limit to the performance of a single disk. Optical disks with larger capacity than magnetic disks may offer high transfer rates in the future, but their access and write times may be slow [Berra et al., 1989].

The traditional approach to improving the performance of a file system is to use several disks, with each file residing on only one disk, to provide a uniform spread of file access requests. This system is suitable for applications making many requests to different files, such as in a transaction processing system, as it reduces the average time that a request waits in a device queue. The method has poor performance if the I/O requests are nonuniform across the disks. It does not help an application that makes infrequent accesses to a single large file, such as in many scientific applications.

The following subsection will describe techniques to improve the raw I/O bandwidth of secondary storage with disk arrays. The next subsection will discuss the techniques used in the Bridge file system for managing data interleaved across disks attached to separate processors. Finally there will be a discussion of the secondary storage requirements derived from the recent research in secondary storage systems mentioned in the preceding subsections.

The organization of the following subsections is as follows:

- Disk arrays,
- Bridge file system,
- and Secondary storage requirements.

4.5.1 Disk Arrays

The large scale production of disks for the personal computer and workstation market has led to the availability of high-density hard disk assemblies at very low cost [Katz et al., 1989b; Katz et al., 1989a]. This has led to

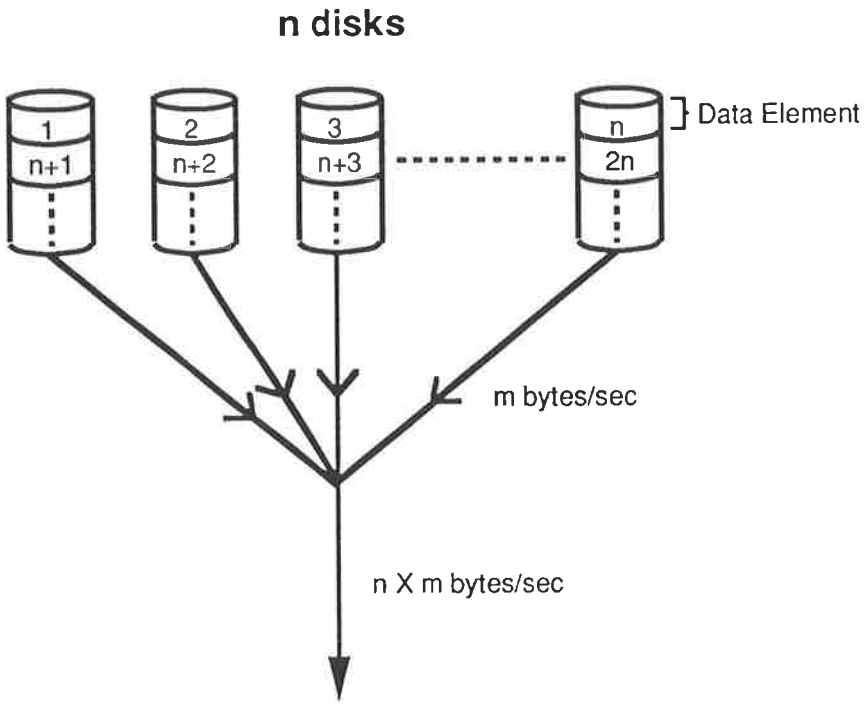


Figure 4.9: Disk array

research in using arrays of disks in parallel to achieve high performance. The basic concept involves interleaving data across the disks (called disk striping), and accessing this data in parallel to provide an increased I/O bandwidth [Salem & Garcia-Molina, 1986]. Figure 4.9 illustrates this concept showing the bandwidth of n disks combined together.

Kim describes the use of a synchronized array of disks with byte interleaving [Kim, 1986; Kim, 1985]. A synchronous motor controls each disk. A central clock, along with a feedback control loop, synchronizes the disks. The technique offers substantial improvements in I/O bandwidth, which suits scientific applications that read large sections of a single file infrequently. It does not provide improvements in the initial seek and rotational latency overheads, hence it will not help applications with many small requests to

different files, where the transfer times are small. Fault tolerance is easy to provide by adding extra disks to allow error correction and continued operation if a disk fails. The disadvantage is the difficulty, and hence expense, of building large synchronous arrays.

Katz *et al.*' have looked at using asynchronous arrays of disks, which allow cheap off-the-shelf drives to operate as an array with a single controller [Katz et al., 1989b; Katz et al., 1989a; Patterson et al., 1988]. The drives independently return interleaved data into a shared buffer. The disadvantage of this scheme is that each transfer incurs the worst case seek and rotational latencies for each access across the group. For large transfers the additional disk latencies will have less effect. The advantage is the saving in cost over a synchronized array.

One disadvantage of spreading each data transfer across several disks is that it does not give any advantage for large numbers of independent small transfers. Patterson *et al.* suggested interleaving data on a disk sector basis (rather than byte), allowing several independent small transfers to occur in parallel [Patterson et al., 1988]. For fault tolerance, they rely on the disk controller to identify which disk has an error, and use an extra disk to allow the recording of parities for correcting the error [Gibson et al., 1989]. Distributing the parity sectors across the disks, prevents one disk becoming a bottleneck in write operations. A single sector read operation only involves one disk. Figure 4.10 illustrates the arrangement of parity blocks among the disks. Note that a write to block 0 of disk 2 can occur in parallel with a write to block 1 of disk 1. A read from block 0 of disks 1 and 2 can occur in parallel.

Reddy and Banerjee have simulated the performance of various combinations of traditional multiple disk systems, byte interleaved synchronized arrays, and block interleaved asynchronous arrays (which they call *declustered ar-*

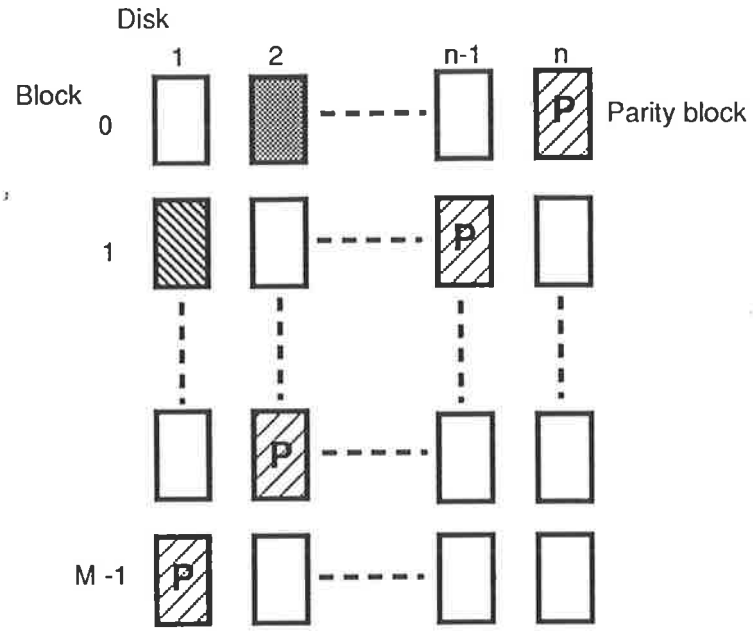


Figure 4.10: Interleaved parity sectors [Katz et al., 1989a]

rays) [Reddy & Banerjee, 1989]. They predict that disk synchronization is impracticable beyond 8 or 16 disks, however the individual disks in a traditional or declustered system could consist of synchronous arrays. They obtained results for both a typical UNIX™ file system workload, and a scientific workload consisting of matrix arithmetic algorithms.

For the UNIX™ file system workload, synchronized arrays gave better results at low request rates, and conversely, declustered systems gave better results at high request rates. Combinations of declustered and synchronous systems worked best with large block sizes.

For the scientific workload, the best performance occurred when all available disks participated in large transfers by having each file spread across all the disks, maximizing the available parallelism. There was little performance difference between the various combinations of synchronized and asynchronous

systems for a scientific workload.

Recent work by Reddy and Banerjee extended the above work to simulate using multiple disks distributed among the nodes in a hypercube multicomputer (with the communication characteristics of a Intel iPSC/2 hypercube) [Reddy & Banerjee, 1990]. They found that when separate processors accessed blocks in a file in random order with the file system workload, the declustered system gave better results even at low request rates. This was because the synchronous array could not take advantage of one processor requesting multiple blocks in a contiguous order thus incurring the disk latency only once.

Reddy and Banerjee also found that with a scientific workload the time to read a file did not scale linearly with the number of disks in the array [Reddy & Banerjee, 1990]. This was because the transfer time across the interprocessor communication links became the bottleneck rather than the disk transfer time.

Olson has also shown that disk arrays can improve the performance for random I/O environments, such as transaction processing and file servers [Olson, 1989].

Parallel arrays of disks can offer substantial performance improvements for a file server. A high speed network allows the file server to provide the benefits of increased I/O bandwidth to processors on the network. In a multiple processor environment, however, the file server may become a bottleneck when many simultaneous requests for data occur.

The approach used in Parallel Goalie was to allow each processor to access its own local files on local disks (see Section 2.2.5.3). This gives the programmer the task of data partitioning and organization for each application. Consider a design verification task that begins by extracting an electrical circuit from a mask layout, and then simulating this circuit for different input combinations.

The optimal partitioning of data for these tasks is usually different. Running multiple tasks on the same set of initial data may require a rearrangement of data for each task. Predicting the optimal arrangement of data before running a task is often difficult, leading to load balancing problems.

The following subsection discusses the Bridge file system, which simplifies the problem of organizing the data into local files by interleaving the data across all the disks in the system.

4.5.2 Bridge File System

The Bridge file system provides an environment for managing data on a multiple processor system [Dibble et al., 1988; Dibble & Scott, 1989]. It interleaves blocks of data among storage devices attached to separate processors. A local file system (LFS) runs on each processor that has an attached storage device. For high performance, these storage devices can consist of arrays of disks operating in parallel. A Bridge Server forms the interface between the file system and user programs. Users have a choice of three interfaces: accessing data through standard sequential operations using the Bridge Server to forward requests transparently to the appropriate LFS; using the Bridge Server to transfer data in a single parallel operation to a group of processes working on one task; or using user defined tools to interact with the LFS level of the system, after obtaining details from the Bridge Server. Tools can create processes on the storage nodes to preprocess information before sending it across the network to the requesting processor.

The assignment of interleaved blocks to disks depends on the application. In scientific applications, where applications typically access files sequentially, a round robin scheme is suitable. For transaction processing, however, when a transaction deletes a record in the middle of the file, all the following blocks would need to move. Bridge supports a linked list representation for block

distributions other than round robin.

The implementation of the Bridge Server depends on the frequency of requests to the server. Where most of the compute intensive processing tasks use tools to interface with the LFS level of the system, programs only request information from the Server when they first access files. This implies that a centralized server would not become a bottleneck [Dibble et al., 1988].

A disadvantage of the system is that a disk failure will corrupt all the files. Dibble and Scott suggest that the improved reliability of storage arrays, via the use of extra disks for error correction, will ensure overall reliability [Dibble & Scott, 1989].

Reddy *et al.* have analysed the problem of embedding I/O nodes into a network of processing nodes, such that each processing node is adjacent to at least one I/O node, which is a processing node with I/O facilities [Reddy et al., 1988; Reddy & Banerjee, 1990].

4.5.3 Secondary Storage Requirements

The two main options for an I/O system suitable for a multicomputer architecture for CAD of circuits are:

- a centralized high performance file server,
- or a file system distributed among the processing nodes.

A centralized file server is the simplest option. It has the advantage of making the use of synchronized storage arrays more cost effective, by centralizing the storage costs. For the compute intensive CAD tasks, such as circuit extraction, sequential file access to large blocks of data is most common. For low numbers of disks a synchronized organization offers the best performance. For large numbers of disks, groups of synchronized disks operating

as a declustered array are necessary. A round robin arrangement of blocks in a declustered array suits the sequential access patterns of CAD tasks. For a parallel processing system where the processing nodes make many simultaneous I/O requests, however, especially for applications that need to store intermediate data, the file server will become a bottleneck despite its high I/O bandwidth.

A system using a file system distributed among the processing nodes, as in the Bridge file system, avoids the bottlenecks of centralized file system software. The use of standard disks (with bandwidths about 3 Megabytes/sec) would keep the costs down, but for better performance the storage devices should consist of arrays of disks. Asynchronous disks appear to offer the best cost trade off between a centralized disk system using expensive synchronous disk arrays, and a distributed system of cheap standard disks. Processes local to the storage nodes can preprocess data, by extracting only the required information, to reduce the interprocess communication overheads. Processing nodes with attached disks would need to allow I/O requests to interrupt the current process to allow local I/O processing. Alternatively, an attached I/O processor could manage the disk, as in the arrangement discussed by Reddy and Banerjee [Reddy & Banerjee, 1990]. Ideally in a system of 20 processing nodes, each node would have an attached storage device to ensure maximum parallelism. For a crossbar interconnection the placing of fewer storage devices than processors is not important. The operating system software can initially allocate processes to processing nodes without storage to try to overlap storage operations with processing.

Figure 4.11 illustrates the arrangement of disks in the system. The workstations will use their own disks with a file system supported by an industry standard operating system. The workstation disks may be local to each workstation (as Figure 4.11 shows), or they may be at some central file server. This file server could be on either a local network connecting the workstations, or connected directly to the crossbar switch with a high speed (e.g 800 Mb/s) link.

The workstation disks will store the master copies of the design description and copies of the application software for the pool of processing nodes. For non-interactive tasks, the system downloads the necessary programs and data to the processing nodes and their disks. A series of applications can work on the data stored amongst the pool of processing nodes without sending intermediate results back to the workstation disks. For example, a user could interactively design a cell, specify parameters of some cells ready for automatic generation, and specify the interconnection requirements of the cells. The user can send this information to the pool of processing nodes, which can do cell generation, cell placement, routing and compaction, using multiple processors as necessary. If the user is happy with the area of the generated design, the pool of processing nodes could do design rule checking and simulation tasks. After a processing session on the pool of processors, the system updates the data stored on the workstation disks. Other designers will access the consistent data base on the workstation disks.

If a fault occurs on a processing nodes, or on a disk amongst the processing nodes, the system can reconfigure to avoid the fault. The user, however, will need to rerun the applications in progress at the time of system failure. The overhead required to ensure that an application can continue where it halted would not be cost effective. Computer aided design tasks do not need the fault tolerance necessary for a transaction processing system (such as a system handling bank accounts).

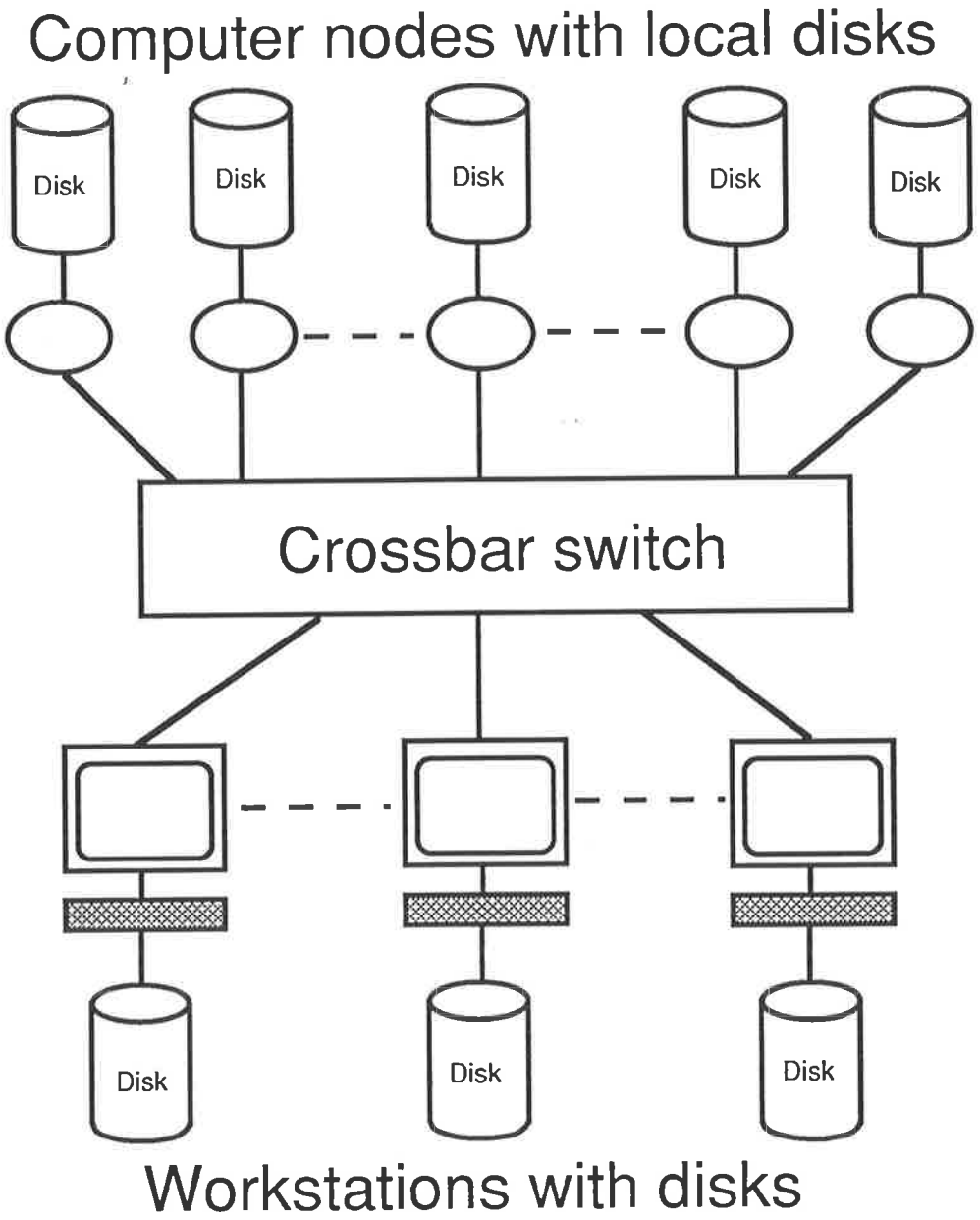


Figure 4.11: Secondary storage architecture

Again to take advantage of bandwidth improvements, large data transfers are necessary to spread the software latencies over a larger amount of data. High bandwidth links ensure that the data transfer between the disks attached to the storage nodes and the workstation disks does not become a serious bottleneck.,

4.6 Operating System

This section will briefly review two prototype distributed operating systems in advanced stages of development: the Amoeba distributed operating system and the Mach operating system. It will then discuss aspects of these systems suitable for CAD of integrated circuits.

With the trend towards distributed workstation computing environments, there has been much incentive for research into distributed operating systems. Tanenbaum and van Renesse have written a good overview of the key concepts of distributed operating systems [Tanenbaum & Renesse, 1985]. They define a distributed operating system as one that looks to its users like an ordinary centralized operating system, but runs on multiple independent processors. The distributed operating system manages the available computing resources automatically. This is distinct from a network operating system that allows users to access files and processing facilities on other computers on a network, but makes the user aware of the location of files in the network and makes the user choose a processor to run a particular application.

Most of the research on distributed operating systems has aimed at general purpose uses of computers. such as text editing and program development, rather than at scientific applications that use multiple processors on one problem. A typical use of a distributed operating system would be to split up automatically the task of compiling several modules of a program, among

several processors. This is in effect multiprogramming, rather than multiprocessing. There is a need to provide programming facilities that allow a software developer to take advantage of parallelism provided by the underlying hardware for a single application. The operating system can provide low level support to implement these facilities, such as providing for interprocess communication.

The rest of this section consists of the following subsections:

- Amoeba,
- Mach.
- and Operating system requirements for the CAD computer system.

4.6.1 Amoeba

The Amoeba distributed operating system [Mullender et al., 1990; van Renesse et al., 1989b] supports a hardware architecture consisting of

- a processor pool,
- workstations,
- specialized servers,
- and gateways connected to a local area network.

The pool of processors, consisting of single board computers, provides most of the computing power. The workstations behave as terminals, and only execute processes that interact intensively with the user, such as window managers that accept commands from a keyboard and a mouse. These workstations could be X-terminals designed to run X Window software [Scheifler & Gettys, 1986]. The specialized servers provide access to dedicated resources

such as disk drives. The gateways provide access to computing facilities located large distances away.

This is a similar architecture to the one this thesis proposes for designing integrated circuits. Thus some of the techniques used in Amoeba may be applicable to the operating system needed for integrated circuit design.

Amoeba is object based, and uses a client server model to operate on these objects. Examples of objects include memory segments and files. A server process can apply operations to an object on a client process's behalf. These operations must be in an object's predefined set of allowable operations. A process must possess a *capability*, which is a ticket specifying the object operations the ticket holder has access to, for each object it wishes to access. Client processes use *remote procedure calls* to send requests to servers for operating on objects [Birrell & Nelson, 1984]. Amoeba determines the server to send the request to from the contents of the capability used to access the object. The client need not be aware of the location of an object in the network. A directory service provides a mapping between ASCII object names and their corresponding capabilities.

The Amoeba designers have kept the kernel as small as possible. User level processes do most operations, making the system flexible. For example users can use their own file systems. The Amoeba kernel handles

- the sending and receiving of messages,
- the scheduling of processes,
- I/O,
- and low level memory management.

Processes in Amoeba have a segmented virtual address space with one or more threads. *Threads* are lightweight subprocesses, with separate program

counters and stacks, which share segments in the process's address space. When one thread blocks waiting for a remote procedure call to complete, the kernel can schedule another thread to execute.

Processes can explicitly map segments in and out of their address space. They can also map files into their address space to allow *mapped file I/O*. This allows a process to access the contents of a file like any other software data structure without explicitly performing file reads and writes.

Amoeba researchers have written a file server called **Bullet** that uses *immutable* files [van Renesse et al., 1989a]. The file server can only create, read, or delete files. Bullet stores files in contiguous areas on disk. File reads involve reading the whole file into the memory of the processor running the file server. Modifying a file requires reading in the old file from disk, modifying it, creating a new file on disk with the modifications, deleting the old file. The Bullet server tries to take advantage of statistics that show that for general purpose computing, 99% of the files contain less than 64kbytes. The simplicity of file operations, and the use of contiguous areas in memory and on disk, leads to high performance. It is not so suitable for scientific uses such as integrated circuit design (with over 100,000 transistors), where files containing more than 10 Megabytes of data are common.

Amoeba has a UNIX emulation package to allow many of the UNIX™ operating system utilities to run.

The following section discusses the Mach operating system, which the Open Software Foundation (OSF) is using as a basis for their variant of the UNIX™ operating system [Murphy & Voelcker, 1990]. Most applications of the Mach operating system seem to be in tightly coupled shared memory multiprocessor systems, in contrast to the Amoeba distributed environment.

4.6.2 Mach

Mach is an operating system designed to be compatible with the 4.3BSD UNIX™ operating system [Quarterman et al., 1985], with support for both distributed computing and multiprocessing [Russell & Waterman, 1987; Rashid et al., 1988; Young et al., 1987]. The Open Software Foundation is using it as a basis for their version of the UNIX™ operating system, called the OSF/1 operating system [Black, 1990]. It uses virtual memory management techniques based on the Accent network operating system [Rashid & Robertson, 1981].

Mach uses 5 basic abstractions [Rashid et al., 1988]:

- task,
- thread,
- port,
- message,
- and memory object.

A *task* consists of a paged virtual address space in which multiple threads of control are possible. *Threads* are lightweight processes with independent program counters that operate within a task. All threads within a task share the same task resources, such as the task address space and open files.

The Mach kernel uses an object oriented interface to system objects, and uses messages for communication. To apply an operation to an object, an application must send a message to the port associated with the object. *Ports* are communication channels that respond to two primitives: send and receive. A port has only one receiver, but may have multiple senders. Mach uses capabilities to control access to ports, as does Amoeba. A process can

give another process access to a port by sending a message containing the port's capability. Access to most Mach facilities is through remote procedure calls on ports [Birrell & Nelson, 1984].

A *message* contains a typed collection of data objects. It can transfer the entire contents of a task's address space. Mach uses copy-on-write memory mapping techniques to send large messages between tasks on the same processor in a distributed memory system. With this technique data copies only occur when either of the two tasks alters the data in the message. These data copies occur on a page by page basis; the rest of the message remains shared.

A *memory object* is a collection of data managed by a server. A task's address space consists of an ordered collection of mappings to memory objects. A user-level task can create a memory object and service requests. Tasks or threads can access objects through ports without knowing their physical location in the network. Mach treats secondary storage in the same way as other objects in the system, by creating a memory object for the storage. It can map files into the address space of a task, hence providing a single level store. This allows file data to be read from disk directly into a task's address space, avoiding the copying operation from kernel buffers typically used in UNIX systems.

Mach supports sharing of memory between tasks with common ancestors on the same processor. When a task spawns a child task, the child task can either receive copies of data objects, using copy-on-write techniques for efficiency, or it can receive access to shared data.

Young *et al.* describe an example of a shared memory implementation for a distributed network of processors [Young et al., 1987]. They use a shared memory server to implement shared memory. The server allows multiple read copies, but when a write occurs to one copy, the server sends an invalidation

message to the other copies.

The Mach designers have kept the kernel as simple as possible, in common with Amoeba. The kernel is a multiple threaded task, which acts as a server that implements tasks, threads, and memory objects. When the kernel creates a new task, thread, or memory object for a client, it returns access rights to a port representing the new object. When a page fault occurs while accessing non-memory resident memory object data, the kernel fault handler sends a message to the port for that object, requesting the data from secondary storage. The server process associated with the port can be user defined. The kernel effectively acts like a cache manager using physical memory as a cache for the contents of memory objects.

A recent paper by Black discusses techniques for scheduling tasks on a shared memory multiprocessor in a time sharing multi-user environment [Black, 1990].

Recent versions of the UNIX operating system such as SunOS also include more advanced virtual memory management facilities such as copy-on-write memory mapping, and the ability to map files into the address space [Gingell et al., 1987].

4.6.3 Operating System Requirements

The computer system this thesis recommends for CAD of integrated circuits has the following resources:

- multiple high performance processing nodes,
- special purpose nodes such as hardware logic simulators,
- multiple workstations for interactive use,
- high performance interconnection network,

- and distributed secondary storage.

These resources are similar to those of the model used for the Amoeba operating system.

The computer system needs an operating system to manage these resources.

Existing workstations will have their own reliable industry-standard operating system, such as a variant of the UNIX™ operating system. Most of the user interactive activities, such as editing and graphical entry of design details, will occur on these workstations.

The high performance nodes need a simple operating system kernel to manage memory allocation, process creation, process scheduling and interprocess communication. Limiting the kernel to essentials will allow for the greatest flexibility. Additional facilities such as a file system can run as user processes. Software developers can develop and debug these user processes without affecting the kernel.

Typically each processing node will be dedicated to a single user, with the user running a compute intensive application. This application will usually involve using other nodes in parallel. Exclusive access to a set of processing nodes ensures that an application can predict an optimal load balance. It also minimizes the response time for a user waiting for the results of an application, thus maximizing user productivity.

The part of an application running on each node may consist of several processes. These processes may be running in independent virtual address spaces, or they may run as subprocesses sharing a single address space. By allowing processes to run in independent address spaces, the system can let software developers develop and run utilities, such as a file system, independently of other application software. By allowing subprocesses of a single application to share a single address space, the system can simplify the shar-

ing of data between these subprocesses. A method to synchronize access to this shared data is necessary, such as the semaphores used in the VORX system (Section 2.2.5.1). The operating system can keep minimal state information for each individual subprocess allowing for fast context switches between subprocesses (or threads), as implemented by operating systems such as Amoeba, Mach and VORX.

The kernel for each node must schedule the processes and subprocesses running on the node. For a single user environment, time-slice scheduling, which gives a reasonable response time for independent processes, is unnecessary. The kernel only needs to schedule processes from a user selectable priority. For example, if a node has an attached disk drive, a file server process may have higher priority over the application process currently running, to prevent processes running on other nodes from waiting too long for disk data. While a process or subprocess waits (blocks) for a response from a process on another node, the kernel will schedule another, possibly lower priority, process for execution.

An operating system kernel that allows software developers to develop software without considering the exact amount of primary memory available, will lead to improved portability of the software. It will allow programs to run on nodes with different primary memory sizes. The kernel can provide memory management facilities to use secondary storage as part of the memory hierarchy available to a program. The primary memory can act as a cache for secondary storage. Applications will be able to handle large problem sizes by allowing the memory usage to dynamically increase in proportion to problem size.

The system may distribute secondary storage among the nodes in the system, with data interleaving, for higher performance. User processes can service requests for data, as allowed in the Amoeba and Mach operating systems.

When a page fault occurs, the kernel can direct a request to a user process to retrieve the data from secondary storage. The user process will decide where to get the data from. The kernel can also send stale pages to a user process to store on disk.

The operating system can support memory mapped files, as supported in Amoeba and Mach, where a process can request the kernel to map file data on disk into their virtual address space. This allows a program to access the contents of a file in the same way as any other data structure. A user process can transfer the data from disk directly into the program's address space.

To reduce the overhead of sending data between processes sharing a processor with independent address spaces, the kernel can use copy-on-write memory mapping techniques, as Mach uses, to avoid unnecessary copying of data.

Processes need access to communication primitives for interprocess communication. The kernel can implement low level primitives such as send, receive, and remote procedure call, such as Amoeba and Mach provide. The send and receive primitives normally are synchronous so that they wait until the other process responds. The kernel can also provide a *Multicast* primitive to provide an efficient implementation for sending a message to a list of other processes [Katseff. 1987]. With some communication networks, separate methods can substantially improve the performance of a multicast relative to several independent send operations. By allowing users access to the network interface, as allowed in VORX, user programs can use their own primitives for special purpose applications. The performance of the software implementing the communications primitives has a major effect on the system performance (Section 4.4.1).

Processes that run on workstations under control of their local operating system can use the same software communication interface, to allow interaction with processes running on the high performance nodes.

4.7 Parallel Programming

Parallel software development would be simple if compilers could automatically extract parallelism from a program written for a serial machine in a conventional serial language. However, the algorithms that work best on multicomputers are often different from the algorithms that work well on a serial processor. This implies that new programs should express parallelism explicitly [Carriero & Gelernter, 1989b]; Carriero and Gelernter discuss techniques for writing parallel programs [Carriero & Gelernter, 1989a]. The aim of current research is to make this task as simple as possible. Compilers are still useful, however, for trying to extract some parallelism in existing serial programs.

Traditionally, tightly coupled multiple processor systems, which have access to a shared memory with one machine instruction, have used a shared variable parallel programming model. In contrast, loosely coupled systems, which send data between processors on a network via messages, have used a message based programming model [Howe & Moxon, 1987b; Karp, 1987; Ranka et al., 1988].

Leler has compared programming parallel machines to programming in assembly language [Leler, 1990]. Parallel programs have been hard to write, since the programmer had to deal with the hardware details, such as interconnection topology. The resulting programs are not portable between systems, removing the incentive for software developers to develop parallel versions of compute intensive CAD software and other applications.

The remainder of this section will begin by comparing the message passing and shared variable programming paradigms. Next, it will discuss three approaches to supporting shared data structures in a distributed memory environment. This section will then briefly mention research into computer-

aided programming tools. It will conclude by discussing the programming facilities needed for the system.

The rest of this section consists of the following subsections:

- Message versus shared variable based programming paradigms,
- Shared variables in a loosely coupled environment,
- Linda,
- Distributed shared virtual memory,
- Shared data object model,
- Computer-aided programming,
- and Programming requirements for the CAD computer system.

4.7.1 Message versus Shared Variable Programming Paradigms

Bal and Tanenbaum give a good overview of the characteristics of the two programming paradigms [Bal & Tanenbaum, 1988].

When a message transfers data between two processes, they must both exist and the sender must know the identity of the receiver. In contrast, data in a shared variable is accessible to any process knowing the address of the shared variable. Processes do not have to exist at the same time, and a process placing data in a shared variable does not need to know which other processes will access it. This also allows the use of *persistent* data which can exist outside the process that created it.

In a tightly coupled system, modifying a shared variable has immediate effect. In a distributed system, however, the nondeterministic delays of several

processes sending messages at the same time makes it hard to determine what order a process will receive messages from several processes.

Message based programming can help make programs more reliable and easier to debug, since processes explicitly send messages to those processes who should accéss the data. Whereas with shared variables there is the possibility of corrupting other processes by bad pointer references, or through synchronization problems [Leler, 1990].

Passing a message between two processes also synchronizes them. In *synchronous* message passing, the sender waits for acknowledgement from the receiver. In *asynchronous* message passing, the sender can continue processing after sending the message, but a receiver must wait until it has received a message before continuing. Shared variable programming requires separate mechanisms to synchronize the activities of processes. Mutual exclusion prevents simultaneous accesses to the same variable. In conditional synchronization, a process waits until a condition is true. For example, a process may wait for a semaphore to become clear before doing something [Dinning, 1989].

In a loosely coupled system, passing large complex data structures between processes executing on different nodes is difficult. For example, a process must pass parameters in a remote procedure call by value, which is inefficient for large structures, rather than passing a pointer to the parameter data structures. This also makes it hard to migrate processes between processors to balance the processing load, because a processor must copy all the process information to another processor.

The next subsection introduces the problem of supporting shared variables in a distributed environment.

4.7.2 Shared variables in a Loosely Coupled Environment

To ease programming in a loosely coupled environment, researchers have investigated ways to simulate a shared variable environment. Bal and Tanenbaum discuss recent attempts to provide some of the advantages of shared variable programming in a loosely coupled system [Bal & Tanenbaum, 1988].

The simplest technique to implement shared variables in a distributed system is to access a single copy of the data, stored on a processor in the network, with a remote procedure call. However, this will be inefficient if processes make frequent requests to the same variable across the network. The alternative is to replicate the shared variable and store it in local memory. The problem comes when a process modifies the shared variable, causing inconsistent copies of the variable to exist.

Some programming environments rely on the programmer to explicitly maintain consistency among multiple copies, such as in a standard message passing programming environment. Programming languages such as Ada allow inconsistent copies of data to exist between synchronization points. Recent systems let a programmer use a shared variable programming environment with operating system support to maintain consistent copies.

The following subsections discuss three different approaches to supporting shared variables.

4.7.3 Linda

The Linda parallel programming paradigm provides an abstraction for sharing data called a *tuple space* [Ahuja et al., 1986; Carriero & Gelernter, 1989b; Leler, 1990]. Tuple space acts like an associative memory. Each tuple consists

of a typed collection of data. It is architecture independent, thus allowing portable programs. There are four fundamental operations:

out Place a tuple into tuple space.

in Match a tuple and remove it from tuple space.

rd Match a tuple and return a copy of it from tuple space.

eval Place an active tuple into tuple space.

These operations easily add to the definitions of established languages, such as the C Programming Language, to support parallel programming. When a process wishes to send data to another process, it creates a tuple for the data, and adds the tuple to tuple space. This is a non-blocking action, similar to an asynchronous message send. A process wishing to receive the data creates a template for the tuple, usually using the first field as a key to search on and using the other fields to specify the expected data types. The process then reads data that matches the template in from the tuple space. If the process finds no matching tuple, it blocks waiting until one is present. There is no direct communication between processes. To create new processes to execute concurrently, a process can create an active tuple and place it in tuple space. The active tuple will do some task and produce a normal passive data tuple as a result. Tuples created by a process can still exist after the process terminates.

To modify shared data in Linda, a process removes the corresponding tuple from tuple space, preventing other processes from accessing it. After modifying the data, the process places a new tuple into the tuple space.

A Linda system consists of a run time kernel for implementing interprocess communication and process management, and a preprocessor for optimizing accesses to tuples at compile time [Ahuja et al., 1986]. The implementation

depends on the underlying architecture. For example each node could store a complete copy of the tuple space. An out operation and an in operation would require broadcasts to all the processor nodes, whereas a rd operation would only need a local search. Alternatively, an out operation could place a tuple only on the local processor node. An in operation would require broadcasting the template to all other nodes and allowing each node to do its own search through its portion of the tuple space. Another technique uses a hashing function on the first field of the tuple to choose which node to send a new tuple, and where to conduct a search for an existing one.

Ahuja *et al.* discuss the architecture of the Linda Machine, which contains hardware support for tuple space and the Linda primitives [Ahuja et al., 1988; Krishnaswamy et al., 1988]. They stress the importance of providing hardware support for the high level abstraction provided by Linda. Leler discusses the use of Linda in the QIX operating system [Leler, 1990; Merrow & Henson, 1989].

4.7.4 Distributed Shared Virtual Memory

Li has developed the concept of a shared virtual memory that allows multiple processors on a network to share a single address space [Li, 1986; Li & Hudak, 1989]. Li implemented an experimental version called **IVY** on an Apollo DOMAIN system consisting of a collection of processors interconnected with a ring network.

A parallel program can consist of a set of lightweight processes sharing a single virtual address space, where the cost of context switching and process creation is small. Some systems, such as VORX [Gaglianello et al., 1989; Gaglianello & Katseff, 1985], support lightweight processes on a single processor, but Li's system allows them to run on separate processors in parallel. Processes can access shared data, distributed in the network, as though it

were on the same processor. This is unlike most distributed systems, where processes running on separate processors have independent address spaces, and access shared data through separate primitives, such as in Linda, or via remote procedure calls.

The system provides an event count primitive for synchronizing access to shared data. A process can advance an event count, or wait until the event count reaches a certain point.

Li's system divides the shared virtual memory into pages. When a process accesses a page which is non-resident in local memory, a fault handler retrieves the page from either secondary storage, or from the local memory of another processor.

Several processors can have private copies of read-only pages. If a processor needs to modify a read-only page, a write page fault occurs. This causes other processors to invalidate their copies, thus maintaining coherence. Only one copy of a page with read-write access can exist. If a process needs to read a page it does not have a copy of, a read page fault occurs. If another processor has read-write access to this page, it changes its access to read-only, and sends a copy to the requesting processor.

Shared virtual memory simplifies the migration of processes between processors to balance the load. It can also remove the separate data-exchange phase found in many parallel applications (such as in Parallel Goalie (Section 2.2), which places a concentrated load onto the communication network. Shared virtual memory moves data on demand as processes access it, thus spreading the communication load over a longer period of time [Stumm & Zhou, 1990].

Li suggests a page size of about 1 kbyte to reduce the possibility of multiple processes requesting access to the same page simultaneously. Most memory management units now support 4 kbyte pages [Milenkovic, 1990]. Larger page sizes can better use the bandwidth of high speed communications. A

processor can use a hardware memory management unit to generate page faults for write accesses to shared pages. Once a shared page is resident for read operations, the cost of access is the same as for local data, unlike other techniques where a program incurs the overhead of calling a routine to handle the shared access every time.

When a read-fault occurs, the system must have a way of finding the page owner. Li allows the page owner to vary dynamically for programming flexibility. Using a centralized page manager to locate a page's owner could create a bottleneck. Li uses a distributed technique that maintains a field, containing the probable owner of a page, for each page in a processor's page table. When a page owner sends a new copy to another processor, it adds the processor to its copy set, which the memory manager uses for the invalidation operation.

Li used four synthetic test programs to evaluate the performance of shared virtual memory. These programs consisted of solving a set of three dimensional partial differential equations in parallel, parallel sorting, parallel matrix multiply, and parallel dot-product. Near linear speedups occurred for the differential equation solution and the parallel matrix multiply, which have a high degree of locality in their calculations. The dot-product program achieved poor results because each the calculation uses each data item only once, and if the data is randomly distributed among the processors, the costs of accessing the data dominate over the computation time.

Li concluded that the performance is good for programs that exhibit locality of data references and as long as most of the references are read-only. Performance falls off for applications that require frequent updates to shared data, or use large amounts of data only once with little calculation (such as a dot-product).

Fleisch and Popek have performed similar research using 3 VAX 11/750s

connected via Ethernet. They experimented with using a window of time over which a processor writing to a page can maintain exclusive access to a page independent of read requests from other processors [Fleisch & Popek, 1989].

Tam *et al.* provide further discussion of distributed systems that support a shared address space [Tam et al., 1990]. These systems allow processors to access data on other nodes by address, as in a shared memory multiprocessor.

A recent paper by Wu and Fuchs describes techniques for allowing a system to recover from hardware faults without restarting the whole system and having to rerun applications with a long computation time [Wu & Fuchs, 1990].

4.7.5 Shared Data Object Model

Bal *et al.* have developed a *shared data object* model that they have used as a basis for a parallel programming language called **Orca** [Bal & Tanenbaum, 1988; Bal et al., 1990]. They have used Orca to develop parallel application programs for distributed systems running the Amoeba distributed operating system [Mullender et al., 1990]. Processes may only access the data in a data object through a set of predefined operations. Data objects shield processes from both the implementation details of the arrangement of data in the object, and the code used to operate on the data. The operations appear as high level primitives to a process. Programs can address and modify the data objects directly as for traditional shared variables.

Orca expresses parallelism by allowing single threaded processes to create single threaded child processes using a *fork* operation. A process can choose between having a child process run on another processor, or having it run on the same processor. A process can pass any of its data objects as shared

parameters when creating a child process. If a process changes a shared data object, all other processes sharing the object see the change. Sharing of objects is only possible between a parent process and its descendants.

All operations on objects occur indivisibly, hence if two processes apply an increment operation to an object, the result will be as if the increments occurred sequentially. This provides for mutual exclusion of a shared data object. Operations on objects can block until an internal condition becomes true, thus providing conditional synchronization.

Bal *et al.* have investigated various implementations of the shared data object model [Bal & Tanenbaum, 1988; Bal et al., 1989a].

Processes running on the same processor can read from a shared data object directly. To prevent processes running on another processor from having to execute a remote procedure call across the network to read a shared object, a run time system creates local copies of the shared object. Instead of always creating a local copy, a run time system can use compile time and run time statistics to decide when to create a local copy. The disadvantage of local copies is that when a process changes a local copy, it must inform all other processors of the change, so that they all see a consistent shared data object.

There are two approaches to propagating changes to distributed copies of shared data objects:

- invalidate all copies of the data except one,
- and update all copies.

Invalidation has high overheads if processes make small changes to large, frequently accessed shared data objects. After a processor makes a change to the data, the system would need to recreate the local copies as the other processors continue to make frequent accesses. The update method avoids

this overhead at the expense of more complicated protocols to keep the data consistent. The other disadvantage of the update method occurs when a single process executes several write operations to a shared data object, it incurs the update overhead for each write, even though other processes may not execute a read operation during this time.

Bal *et al.* have described an implementation that replicates each data object on all processors with processes that access it, and updates local copies by applying write operations to them all [Bal et al., 1989a]. The Orca language implementation consists of a compiler and a run time system. The run time system runs an object manager process on each processor. This manager process is responsible for altering the contents of data objects. The manager process shares the part of the address space that stores data objects with processes on the same processor. The compiler generates a call to a run time system routine for each time a program applies an operation to a shared data object. If it is a read operation, the run time routine calls the routine that implements the read operation in the object definition. The read operation accesses the object directly through the shared address space. If the operation requires altering the contents of the data object, the run time routine sends a message to all object managers to update their copies of the shared object. The user program blocks while waiting for the local object manager to update its local copy.

The shared data object implementation described above works well for applications that require frequent reading of a shared variable. by multiple processes in parallel, with occasional updates. It would be suitable for low temperature, parallel simulated annealing, where the algorithm tests many changes to a shared circuit description, but only accepts a few of them. Bal *et al.* point out that providing the high level abstraction of shared data results in a loss in efficiency compared to message passing programming. Parallel programs will perform better if they minimize the accesses to shared data,

and do most of the work with local data.

A recent paper by Stumm and Zhou [Stumm & Zhou, 1990] analyses algorithms for sharing data in a distributed system, similar to those used in Li's shared virtual memory [Li & Hudak, 1989] and Bal's shared data object model. Stumm and Zhou compare the advantages and disadvantages of multiple-readers/single-writer replication (where a writer invalidates copies of data on other processors) and multiple-readers/multiple-writers replication (where each writer sends updates to shared data on other processors). They concluded, as expected, that the performance of each scheme depended on the memory access characteristics of a particular application. They state that many applications have their input data widely read-shared by processors, and have updates to a particular region of data made by a single processor (hence little write contention). Thus multiple readers/single writer replication seems a good solution.

Stumm and Zhou recommend that application developers have available a choice of which distributed shared memory scheme they use according to their application [Stumm & Zhou, 1990].

The next subsection will briefly introduce some recent research into computer-aided programming tools for easing the task of writing parallel programs.

4.7.6 Computer-aided Programming

Many of the ideas associated with computer-aided design are applicable to computer-aided programming. The aim is to increase the productivity of a software developer by automating the simple and repetitive tasks, and providing facilities to manage the complexity of the program. With the variety of parallel computer architectures and the difficulty of debugging programs with errors, the development of computer-aided programming tools

will be essential to the acceptance of parallel programming.

This subsection will briefly overview two approaches to computer-aided programming tools: CAPER and Hypertool.

4.7.6.1 CAPER

Sugla *et al.* have described the CAPER Concurrent Application Programming Environment that takes a building block approach to developing parallel programs [Sugla et al., 1989] on the message based HPC (described in Section 2.2.5.1). The approach is analogous to the use of standard cells in circuit design, where the reuse of highly optimized cells saves design time. Sugla *et al.* mention the development of a VLSI design rule checker as one application of the environment.

CAPER consists of a working set of commonly needed parallel algorithms (such as sorting algorithms), and a set of data transformation routines. The data transformation routines allow data output from one parallel algorithm to be rearranged (and possibly redistributed among multiple processors) into a form suitable for another parallel algorithm. CAPER also provides a mechanism to support shared data by the creation of a shared-data manager process.

Programming in CAPER can consist of using a graphical interface to draw a flow graph of the program consisting of parallel algorithm blocks interconnected with data transformation blocks. The system automatically sets up the communication channels between the blocks of software. Software developers can add their own sections of either sequential or parallel code to the flow graph. A developer can include program blocks that will monitor the performance of the program when it runs on the target machine, and send performance information to the workstation screen.

4.7.6.2 Hypertool

Wu and Gajski discuss the problems of partitioning a program into sections that can run in parallel, and scheduling these sections so that processors do not become idle waiting for another processor [Wu & Gajski, 1989]. They have developed a tool called Hypertool [Wu & Gajski, 1990] that adopts a partitioning and merging approach to parallel programming.

One approach to partitioning a program is to break it into a number of partitions equal to the number of processes as implemented in Parallel Goalie (see Section 2.2). This is suitable for problems with regular structures but can cause load balancing problems for irregular structures. The other approach is to partition the problem into many partitions and assign sets of partitions to processors. When one partition becomes blocked, a processor can schedule another partition for execution to prevent standing idle.

Hypertool requires a user to develop a parallel algorithm and partition it into many partitions (called processes). Hypertool then creates a macro dataflow graph for the program that shows the dependencies of the partitions. It uses a scheduler to merge partitions into tasks that can run on one processor. The scheduler aims to reduce interprocessor communication and processor idle time, thus minimizing the total execution time. Hypertool assigns partitions that require a large amount of intercommunication to one task. The problem is similar to the problem of partitioning a logic circuit among processors for parallel simulation. The scheduler can use information about the underlying architecture to optimize the merging step. After the merging stage, hypertool adds communication primitives to the code for communicating to processors over a network.

Hypertool provides an estimate of the performance of the parallel program by using a simulator that models the execution of the program on a message based system. The simulator provides information such as the number

of message transmissions, message lengths, the transmission time for each message. and the network delay for each communication channel. An explanation facility can display graphical information showing the flow of data among processors and the load distribution.

Wu and Gájski claim that Hypertool increases programmer productivity by an order of magnitude. By automating the addition of communication primitives, Hypertool reduces communication errors and avoids deadlock. Good scheduling algorithms improve the quality performance of parallel programs. Hypertool also provides for portable code because it handles the translation to machine code.

4.7.7 Programming Requirements

The computer system should support widely used programming languages, such as C (and object oriented C++) and FORTRAN. Software developers can use these languages for developing software for the user interactive applications on the workstations, and for the compute intensive applications on the high performance nodes. They can easily port sections of software already written for other machines.

For explicitly specifying parallel execution of applications on the high performance nodes, there are two choices:

- a new parallel programming language,
- or an existing language with extensions.

Software developers are more likely to develop software for the system if they can use a language that they are already expert in. Well developed sets of standard routines for doing many common operations are available for established languages. Ideally a language tailored to parallel programming

would be best, but there are currently no widely used parallel languages available; Bal *et al.* present a thorough survey of the parallel languages available for distributed systems [Bal et al., 1989b]. Thus the best solution for now, is to provide extensions for an existing programming language. The disadvantage of this solution is the lack of standard extensions, making it difficult to provide software that is portable between different systems.

Software developers will benefit from having available both message based and shared variable programming extensions when developing CAD software. This is because of the wide range of techniques used in CAD applications. A recent paper by Gabber discusses a package of routines for extending an existing programming language that supports both message passing and shared variable programming [Gabber, 1990].

Message based programming benefits from less overheads in communications. Software developers can easily develop many parallel CAD applications using this model. Existing languages can call routines to do the message passing. These routines may be kernel supported, or user developed. The basic routines would include asynchronous send and receive messages. Additionally, remote procedure call routines allow for client/server operations, where a typical server would be a file server. Multicast routines are useful for sending the same message to several processors [Katseff, 1987].

To ease programming for applications that suit a shared data structure model of programming, a software developer should have the option of using a shared variable programming abstraction. The two alternatives are:

- to provide a set of primitives to manage a shared address space (e.g Linda),
- or provide shared memory as part of virtual memory management.

The Linda paradigm offers the potential of portability, by providing the tu-

ple space abstraction, but it has high overheads that may make it inefficient for some programs on architectures without hardware support for the Linda primitives. If Linda becomes widely used in the CAD community, the computer system should support an implementation of Linda to allow software from other systems to run.

By providing the ability to run subprocesses sharing part of their address spaces across distributed processors, the system provides a simple extension of the subprocess (or thread) model used on a single processor (such as in VORX and Mach). Li has shown how this can work using memory management hardware to provide the distributed virtual memory [Li & Hudak, 1989]. When a process creates a subprocess, it can pass the data structures it wishes to share to the routine that creates the subprocess. This is similar to how Bal *et al.* pass data objects to child processes, in their shared data object model [Bal & Tanenbaum, 1988]. The routine that creates the subprocess will execute virtual memory management routines that will mark the pages associated with the data objects as shared pages. When a subprocess attempts to write to one of these pages, the fault handler will take care of memory coherency problems as Li has described.

Unlike in the shared data object model, there is no need to call a routine (handled by the compiler) each time a subprocess accesses a shared data structure. The memory management software handles accesses to data that can affect the coherency. This has advantages when accessing shared data in frequently run inner loops. If the data is read only, or the processor has write access, the process accesses the data as for any other local data.

By basing access to shared data structures on a page by page basis, the system can use memory management hardware to assist in creating the shared memory model. This technique also reduces the chance of contention for the same item of data compared to basing access at the data object level.

An invalidation protocol is suitable because the granularity of data access is smaller. When a write operation occurs to one page, the system only needs to invalidate copies of this page, not the whole data structure. Thus the update protocol used in the shared data object model is unnecessary.

To make the best use of the shared variable abstraction, software developers must be aware of the limitations. Applications should minimize the number of accesses to shared data, and do most of the calculations with local data. The shared variable abstraction works well where processes make multiple read accesses to the same shared data, and where the number of updates to the shared data is infrequent.

CAD algorithms can require access to large data structures, such as a circuit description, beyond the capacity of the entire machine's primary memory. Virtual memory management techniques simplify the task of managing these large data structures. Subprocesses only need to keep pages of data structures that they are currently using in local memory. Often a subprocess will be the only one to carry out a write operation on a particular part of a data structure. The memory management software can support the swapping out of rarely used portions of a large data structure to secondary storage. This allows software to manipulate data structures larger than the total primary memory available on the machine. Clearly performance will deteriorate as the accesses to secondary storage increase with problem size.

As parallel programming becomes more widespread, the common use of message passing and subprocess creation primitives should provide incentive for standardizing the names and characteristics of these routines.

The use of computer-aided programming techniques for parallel machines will improve the productivity of software developers. This will be a major step towards putting parallel programming into the mainstream of software development, by providing for portability of software. These tools require a

good graphical interface, which the system architecture provides using the workstations. A graphical interface is also useful for debugging parallel programs. McDowell and Helmbold survey techniques for debugging parallel programs and point out the importance of graphical interfaces [McDowell & Helmbold, ,1989].

Chapter 5

Conclusions

Recent advances in design tools have enabled designers to effectively manage the complexity of integrated circuits with over 100,000 transistors. Now designers need faster computing systems on which to run these tools, to reduce the overall design time necessary to produce a circuit that meets the requirements of its application. Only by reducing the design time is it possible to make use of advanced integrated circuit technology in low production volume applications.

This research has found that there are a wide range of algorithms with different computational characteristics used in computer-aided design (CAD) of integrated circuits. The wide diversity of these algorithms makes it difficult to find a computer architecture that will be optimum for them all. The thesis concludes that a computing system architecture consisting of a pool of computer nodes accessed through a set of workstations offers the greatest flexibility.

Commercially available workstations are inexpensive enough to allow a designer to have exclusive use of one for running the user interactive design tools. The graphical interfaces of workstations allow designers to enter de-

sign details in a variety of representations, and provide visualization facilities for evaluating the output of design verification tools such as circuit simulators. Exclusive use of a modern workstation ensures that the response time to design input is fast enough to maintain a designer's concentration. The workstations should support an industry standard operating system and graphical interface, to allow a designer to benefit from the wide range of quality design software already available.

The pool of computer nodes can provide designers with a shared resource for running the compute intensive applications such as simulated annealing for logic synthesis, design rule checking, circuit simulation, and fault simulation. These computer nodes can reduce the running time of these applications by cooperating to solve different parts of a problem in parallel.

This thesis has discussed recent work in parallel algorithms for using multiple processors to run CAD tools. It has found that most parallel CAD algorithms use a model where each processor executes calculations independently on separate data, using a small amount of communication with other processors to exchange intermediate results. This thesis thus recommends the use of a pool of computer nodes with independent memories that cooperate by passing messages across an interconnection network. This architecture is simpler and cheaper to build than a shared memory architecture that needs special hardware to support a global shared memory.

The execution time of most parallel algorithms on multiple processors begins to reach a lower limit as the algorithm uses additional processors. For most current parallel CAD algorithms, the factor by which multiple processors reduce the execution time compared to a single processor reaches a limit at around 20. Thus this thesis recommends the use of a few (*e.g.* 20) complex computer nodes rather than many (*e.g.* 1000) simple computer nodes.

To minimize the cost of the system, the computer nodes should take ad-

vantage of high production volume microprocessors. Recent innovations in these microprocessors use multiple functional units and pipelining to exploit instruction level parallelism. These innovations can double the performance over a conventional microprocessor design. The computer nodes need good floating point performance to support tasks such as analog circuit simulation. This performance is available from floating point units integrated into the microprocessor, thus exploiting the wider bandwidth paths and multiported registers available on-chip. The large amount of data needed to describe large integrated circuits places heavy demands on the amount of physical memory. Thus the computer nodes should maximize the amount of local memory to minimize the number of references to secondary storage. This requires the use of high density dynamic RAMS with static RAM caching.

Parallel applications such as logic simulation frequently send intermediate results to other processors during execution. The interconnection network must be fast enough to prevent the communication becoming a bottleneck. Thus this thesis recommends using a crossbar switch with 800 Mbit/sec links for interconnecting the computing nodes. Slower links, such as 100 Mbit/sec, that operate over longer distances would be appropriate for connecting to the workstations. The use of around 20 computer nodes allows the use of a crossbar switch. Larger scale systems that support additional users and other application areas, which can make use of additional processors, could consist of several of these crossbar switches interconnected together.

Secondary storage is necessary to store application programs and circuit description data. Also, when the size of a problem is too large for all its data to remain in the primary memory of the computer nodes, secondary storage must hold the intermediate data. Thus the access time to secondary storage must be low enough to avoid the I/O time from dominating the processing time, and limiting the performance benefit of using multiple processors. This thesis recommends distributing disks among the computer nodes, and

interleaving file data across these disks. This allows high I/O data rates, and distributes the I/O requests across multiple processors to prevent one processor becoming a bottleneck.

The thesis recommends the use of a simple operating system kernel running on each of the processors. The kernel should manage memory allocation, process creation, process scheduling, and interprocess communication. Additional facilities such as file systems can run as user level processes. Processes should be able to run in independent virtual address spaces, or run as subprocesses sharing a single address space to ease the sharing of data.

Good program development facilities are essential to maximizing the potential of multiple processors to cooperate in parallel to reduce the running time of an application. This thesis recommends supporting programming languages widely used in industry such as C and C++. Extensions to these languages, in the form of library routines for creating subprocesses and communicating between processes, can provide access to the parallel processing capabilities of the system. Computer-aided programming techniques for programming parallel machines are an important area of further research. These techniques have the potential of offering similar advantages to software developers as computer-aided design tools have provided for circuit designers. The graphical user interface on the workstations will provide a productive platform for these techniques.

Parallel algorithms for applications such as simulated annealing and circuit simulation favour the use of large shared data structures. A message passing programming paradigm makes it difficult for a software developer to coordinate access to shared data. Thus this thesis recommends providing support for the shared variable programming paradigm, normally found on shared memory machines, at the memory management level. This high level abstraction will result in a loss in efficiency compared to message passing,

particularly if an algorithm makes frequent updates to a shared structure. However it allows for faster prototyping of parallel software.

In summary, this thesis provides a framework for either building a new system or modifying an existing system to reduce the running time of CAD applications through the use of parallel processing. It also provides a survey of the techniques used in parallel CAD algorithms that would assist in developing applications for the system.

Although detailed simulations of the effect of hardware and software parameters of the proposed system on the running time of CAD applications used in developing large integrated circuits would be useful, this would be very time consuming. The alternative, which others have already done, is to make simplifying assumptions and simulate parameters of sections of the system with workloads representative of those used in practical applications. Thus the next step is to develop a suite of industrial quality CAD applications on an implementation of the system, and evaluate the effects of system parameters on the performance of these applications, when used to design integrated circuits with over 100,000 transistors.

Appendix A

Real Time Subsystem Architecture

This appendix discusses a commercial alternative that the Electrical and Electronic Engineering Department looked at before deciding to fund the design and construction of the real-time subsystem. It then goes on to detail the specifications and resulting architecture of the real-time subsystem.

A.1 Commercial Alternatives

The Department looked into buying a commercial laboratory computer to meet its needs, in particular the MASSCOMP[1] MC-500. Figure A.1 shows the architecture of this computer. The design aims to meet the requirements of both general purpose and real-time applications. It achieves this by dividing up the tasks of data processing, data acquisition/control, and data display, among three main processors.

[1]MASSCOMP, RTU, Performance Architecture, STD+Bus, and Triple Bus are trademarks of the Massachusetts Computer Corporation

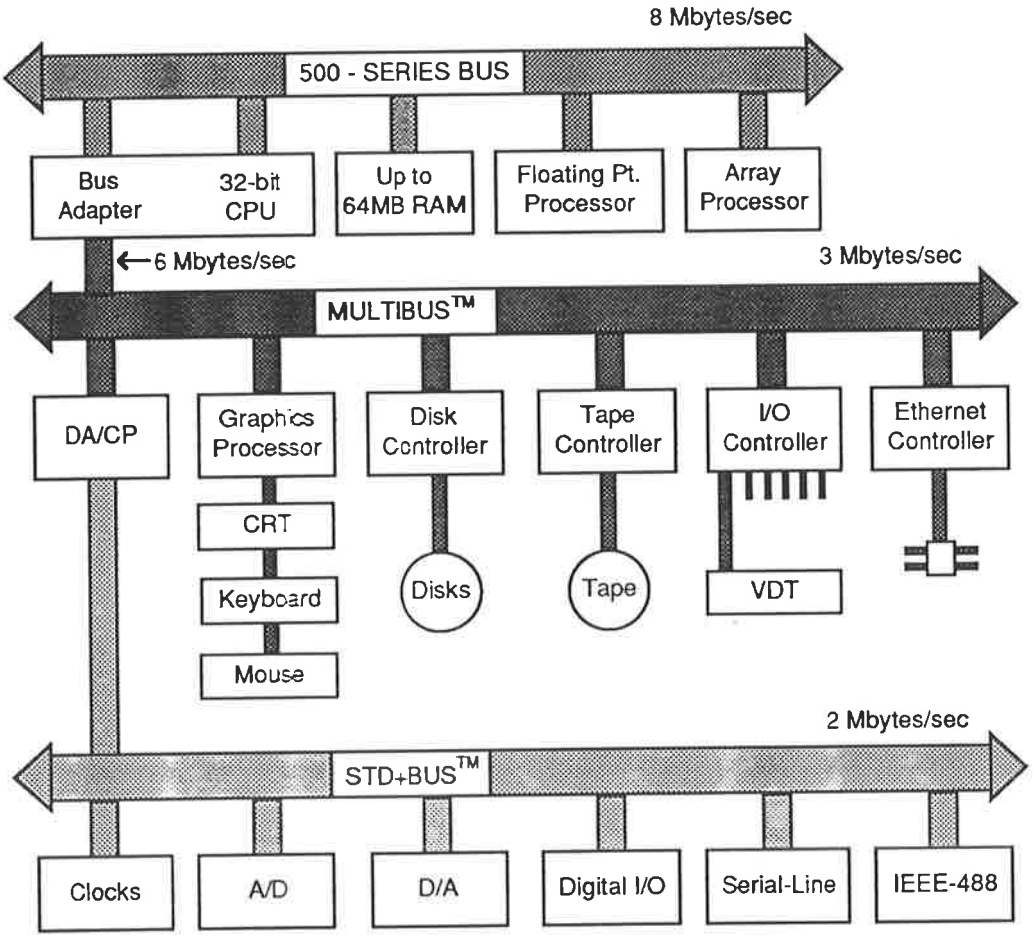


Figure A.1: MC500 Minicomputer Architecture [Masscomp,]

The data acquisition and control processor (DA/CP) attaches to a modified version of the industry standard STD bus. The STD+Bus™ consists of two standard 8-bit wide STD buses connected in parallel to provide a combined bandwidth of 2 Mbytes/sec; this allows analog data acquisition at sampling rates up to 1 million samples/sec with 12 bits of resolution. Bus modules are available to provide analog and digital input/output (I/O), for interacting with real world systems. Interfaces are also available for connecting to other laboratory instruments that often have either a serial line, or General Purpose Interface Bus (GPIB, IEEE-488) connection. The DA/CP is responsible for the management of these facilities, and has direct memory access (DMA) to the CPU's main memory. The main memory serves as a data buffer for the DA/CP, as it transfers data to disk over the 16-bit wide Multibus[2](IEEE-796), which has a bandwidth of 3 Mbytes/sec. The DA/CP provides simple preprocessing of the data before storing it in memory.

Multibus™ provides the interface for system peripherals such as disk drives, terminal controllers and local area network interfaces. It also connects to up to four Graphics processors that provide a window based graphics display for presenting data. These processors are Motorola 68000 based.

The main central processing unit (CPU) attaches to a proprietary, 32-bit wide system bus that has a bandwidth of 8 Mbytes/sec. There is enough bandwidth to allow the CPU to provide general purpose processing tasks while the DA/CP carries out high speed transfers. The CPU uses a 16-bit Motorola MC68010 processor to provide general purpose processing. Optional accelerator cards are available for providing high speed floating point and array processing.

The MC500 uses a modified version of the UNIX operating system, called the RTU™ operating system, to support real-time applications. It allows

[2]Multibus is a trademark of Intel Corporation

fixed priority real-time tasks. The highest priority task gets unlimited CPU time until it completes, pauses, waits for I/O, or a higher priority process interrupts it. A real-time process can prevent the operating system swapping it to disk, or paging parts of it to disk. A task can use contiguous areas on disk to store data. This ensures that the system maintains a predictable, sustained data rate to disk; normally an operating system creates fragmented files to maximize the disk storage available. Blackett [Blackett, 1983] describes the characteristics of the operating system in more detail.

Masscomp designers describe this system in more detail [Boxer, 1983; Cane & Mullen, 1984; Dewey Jr., 1983; Levreault Jr., 1983].

A.2 Specification for the Real Time Subsystem

The Department decided against purchasing a commercial system such as the Masscomp MC500, because it duplicated many of the facilities already available for general purpose computing, and hence was too expensive for the data acquisition and control facilities it provided. Instead, the Department decided to fund the design and construction of a real-time subsystem (RTS), for the VAX™ 11/780 general purpose computer system (GPS), to provide the real-time facilities required for laboratory research. Figure A.2 shows a block diagram for this system.

The initial specifications for the RTS were

- minimal operating system overheads,
- 32-bit processing architecture,
- at least 1 Megabyte of memory,

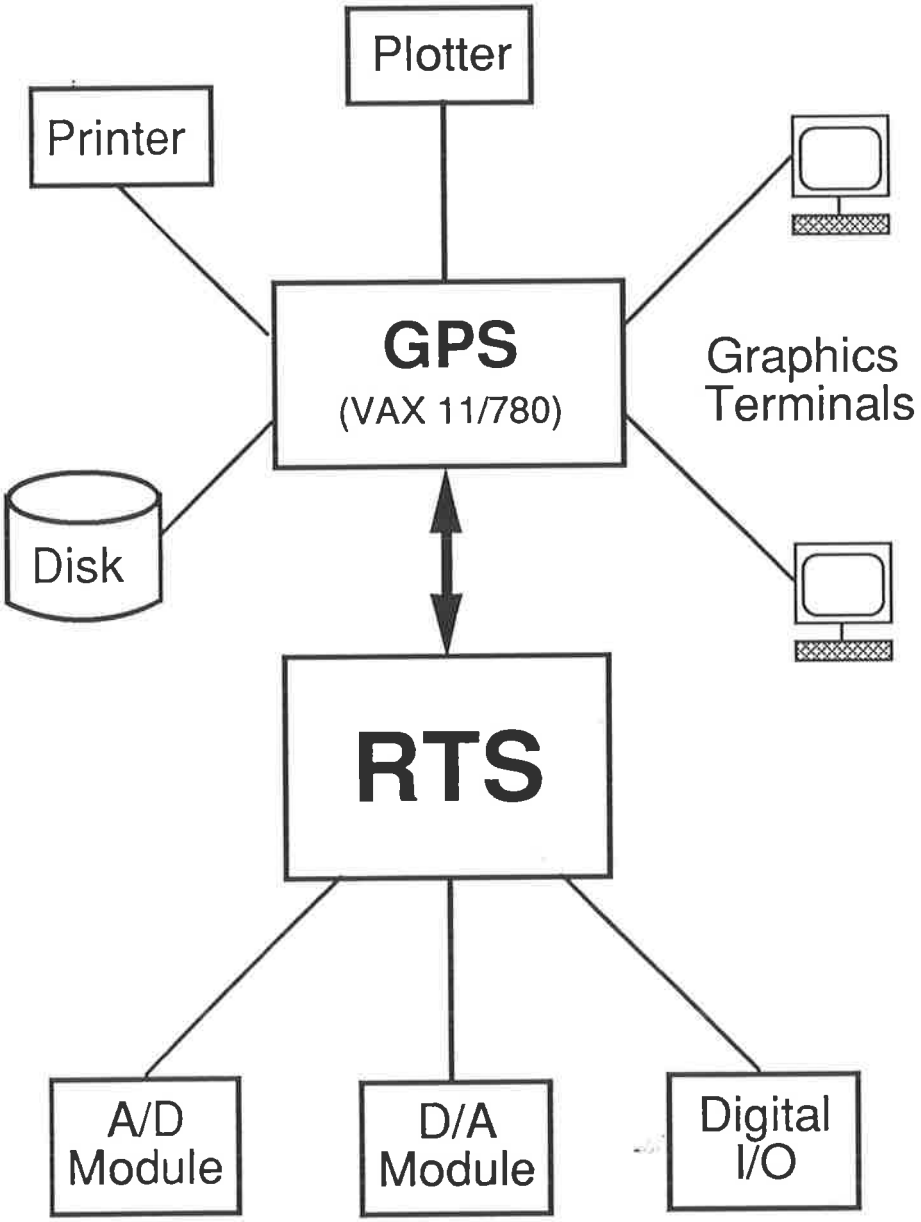


Figure A.2: System Overview

- at least 24-bit address space,
- A/D and D/A facilities with at least 8-bit resolution, 4 channels and 500 kHz sampling rate,
- A/D and D/A facilities with a least 20 kHz sampling rate, 12-bit resolution and 4 channels,
- high speed (200 kbytes/sec) data link to the VAX™ 11/780,
- industry standard bus backplane.

Without operating system overheads, such as file system management, memory management, and process scheduling, the RTS will minimize its response time to external events. The GPS will supply the environment and utilities for RTS software development and analysis of results. As the RTS will be without virtual memory, the available memory of the RTS will limit the size of real-time processes. Only one user at a time will operate the RTS, so the GPS will only load programs, needed for the current experiment, into RTS memory. All these programs will remain resident in the RTS memory for the duration of the experiment; facilities for swapping idle processes out of memory will be unnecessary. The timing behaviour of the RTS will be predictable, allowing it to meet the real-time needs of experiments. A library of routines will be available on the GPS for hiding the complexities of controlling the I/O hardware on the RTS.

The main processor of the RTS shall take advantage of a modern 32-bit wide microprocessor architecture. The microprocessor will provide local processing for real-time control, and preprocessing of acquired data before transferring it to the GPS for further non real-time processing. For real-time control applications, it is desirable to have some floating point processing support, perhaps with a coprocessor chip. The main processor will provide the same

functions as the DA/CP of the MC500 (see section A.1) for controlling the operation of I/O interfaces to the external world.

The RTS needs enough local memory to store user programs, and buffer data during data acquisition. At high sampling rates this memory will fill rapidly; for example, 1 second of 8-bit sampling, at 1 million samples/second, will fill 1 Megabyte of memory. At least 1 megabyte of local memory is desirable, with room for further expansion if necessary. To allow for future expansion, the RTS requires at least 24 bits of addressing, which can address up to 16 Megabytes of memory.

For high speed data acquisition applications, the RTS needs to acquire data at a rate of at least 500,000 samples/second, with at least 4 channels. The higher the sampling rate, the harder it is to achieve a high resolution in the samples. For most applications that require high sampling rates, 8 bits of resolution will suffice.

For applications that need accurate values for I/O, such as some real-time control applications applications, the RTS I/O needs a resolution of 12 bits/sample. At this level of accuracy, a sampling rate of 20,000 samples/sec is acceptable.

The RTS needs a high speed data link to the GPS to receive programs downloaded from the GPS, and to send back acquired data for further analysis. A conventional 9600 baud serial line takes about 15 minutes to transfer a Megabyte of data. This is too slow if several blocks of measurements need to be taken. The data link should allow the RTS to transfer data at the maximum rate that the VAX™ can store data onto disk, which is about 200 kilobytes/sec.

To avoid having to design and build each component of the system, the RTS shall use an industry standard bus backplane. This allows the RTS to take advantage of bus modules from different manufacturers who each specialize

in different areas of module design. Bus modules include analog-to-digital (A/D) converter, digital-to-analog (D/A) converter, digital input/output (I/O), central processing unit (CPU), and memory modules. Modules incorporating analog components are particularly hard to design to achieve high resolution. The object of CPU module design is to cram as many facilities as possible onto the available board area. This maximizes the performance of the CPU as it can does not need to access as many facilities over the bus, but makes the design more complex. Thus the design of different types of modules requires different design skills, and designers benefit from purchasing modules from manufacturers who specialize in particular areas of module design. Purchasing modules also saves time, and hence reduces costs for the overall design.

A.3 Architecture of the Real Time Subsystem

The Real Time Subsystem comprises:

- a 12-slot VMEbus backplane enclosed in a rugged card cage with cooling fans and power supply,
- a single board computer (SBC) module,
- a debug/diagnostic monitor for the SBC module,
- a 16-bit wide data link to the GPS (VAX 11/780),
- a 16-channel high speed analog-to-digital (A/D) converter module,
- an 8-channel high speed digital-to-analog (D/A) converter module,
- and a 32-bit digital I/O and programmable timer module.

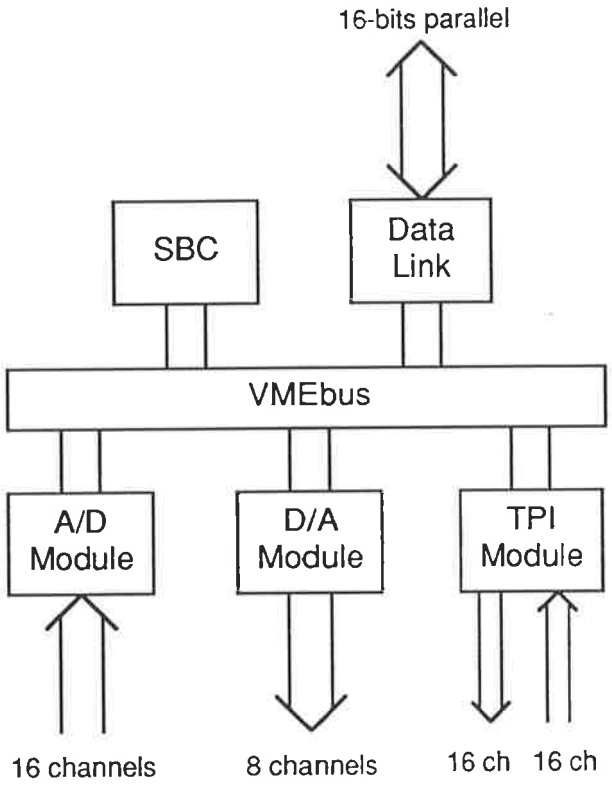


Figure A.3: Real Time Subsystem (RTS) Architecture

Figure A.3 illustrates this architecture.

A.3.1 VMEbus Backplane

The RTS uses VMEbus as its industry standard backplane bus [Micrology, 1985]. VMEbus is one of several 32-bit wide buses that were available at the time of the RTS's design. The other buses were Multibus II, Nubus, and Futurebus. Borrill has written several papers describing the characteristics of these buses [Borrill, 1989; Borrill, 1985: Borrill, 1986].

VMEbus is the most widely used bus in industry, and hence has many bus modules available for interfacing to external systems. It uses an asynchronous

protocol for data transfers across the bus; this allows modules of different speeds to operate together, and allows speed improvements to be made to individual modules without disturbing the rest of the system design. VMEbus modules are of two sizes that adhere to the Eurocard standard (IEEE 1011): the double height, standard depth Eurocard (233.4mm x 160mm) and the single height, standard depth Eurocard (100mm x 160mm). The smaller size allows for a smaller scale bus that only supports 16-bit wide transfers. The bus connectors are reliable, two part connectors sealed against corrosion. The mechanical characteristics of the bus system make it ideal for using in an industrial environment subject to vibration and dust. The maximum quoted bandwidth of the bus is 40 Mbytes/sec [Borrill, 1989].

The wide availability of bus modules for VMEbus, its asynchronous protocol, its rugged construction. and its high bandwidth compared to existing facilities, led to using VMEbus as the foundation for the RTS.

A.3.2 Central Processing Unit

The Motorola MVME-133 single board computer (SBC) module serves as the central processing unit of the RTS [Motorola, 1986]. It consists of

- a 16MHz Motorola MC68020 microprocessor coupled with a 16MHz MC68881 floating point coprocessor,
- 1 Megabyte of local memory,
- facilities for terminal I/O.
- facilities to handle on-board and VMEbus interrupts.
- four 28 pin sockets for ROM/PROM/EPROM/EEPROM,
- four 8-bit timers,

- a real-time clock,
- and a VMEbus system controller interface.

The department already had facilities for doing software development work for the Motorola MC68000 microprocessor on the VAX 11/780, as part of its teaching program. This led to choosing the MC68020, which is object code compatible with the MC68000 processor, as the basis for the 32-bit processing architecture of the RTS [Motorola, 1984].

The MC68020 microprocessor has a full 32-bit architecture. It has 32-bit wide registers and data paths, with 32-bit addressing. The chip provides separate 32-bit address and data buses for interface to external circuits. It has a few added features compared to the MC68000, such as new addressing modes, an on-chip instruction cache, and a coprocessor interface.

The MC68881 floating point coprocessor can speed up floating point calculations performed by the MC68020 by up to 100 times [Motorola, 1985]. The coprocessor fully implements the IEEE standard for Binary Floating-Point Arithmetic. It supports several data formats including integers and single, double, and extended precision real numbers. Before operating on any data, it converts all data formats into 80-bit extended precision format. It supports most mathematical operations including: arithmetic (e.g add, subtract, multiply, divide, and square root), trigonometric (e.g sine, cosine, tangent), and logarithmic (in bases 2, e, and 10).

Both the MC68020 and the MC68881 operate at a clock frequency of 16.67 MHz on the SBC. The MC68020 can access memory in a minimum of 3 clock cycles. The 1 Megabyte of memory on the SBC contains 120ns, 256K $\times$ 1 dynamic RAM ZIPS (zig-zag-in-line package). Access to the memory array requires 4 clock cycles, thereby introducing one wait state for memory accesses by the MC68020.

The on board real-time clock provides the time from tenths of seconds to tens of years. It can interrupt the MC68020 at intervals of 0.1, 0.5, 1, 5, 10, 30, or 60 seconds.

The SBC provides an A24/D32 VMEbus interface. This means that it drives only the least significant 24 bits of the 32-bit VMEbus address bus, whereas it can access all 32-bits of the data bus. In any one address space it can only access 16 Megabytes of memory. Instructions and data can occupy separate address spaces.

The SBC can generate an interrupt to other modules on the VMEbus. The on-board interrupt handler services and arbitrates all 7 priority levels of interrupt on the VMEbus.

The SBC can request control of the VMEbus if another module is currently in control. An on-board single level arbiter arbitrates requests from modules requiring control of the VMEbus. It relies on the VMEbus bus grant/bus request daisy chain to do the arbitration.

The SBC also provides the 16 MHz VMEbus clock, and a system reset signal that will reset all other modules on the VMEbus.

These facilities provide a powerful system for processing of data before sending it to the GPS, or for real-time processing of data.

The SBC acts as the bus master on the RTS; all the other modules now in the system behave as bus slaves. They can either contact the bus master by generating an interrupt, or the master can standby and poll them to determine when they need to be serviced. Researchers can alter the priority of the interrupts to suit each application. The other bus modules use a simple VMEbus interface, which allows more room on the slave modules for extra functions.

A.3.3 On-board Software

The MVME-133 single board computer module has an on-board monitor program stored in ROM that provides

- an interface to a standard RS-232-C ASCII terminal, which connects to the front panel of the SBC to act as a system console terminal,
- facilities for loading and executing user programs under complete operator control,
- commands for displaying and changing memory,
- EEPROM (Electrically Erasable PROM) programmer,
- hardware diagnostic and debug commands,
- a one-line assembler/disassembler for program monitoring and changing, with support for the floating point coprocessor,
- self test power up,
- terminal I/O routines callable from user written programs,
- and a transparent mode for connecting the console terminal through to remote host connected to a rear serial port.

Additional code is available in ROM to control the communication over the data link to the VAX 11/780. The EEPROM programmer aided the development of this additional code. The monitor is the only operating system facility available on the RTS.

A.3.4 A/D Converter Module

The Burr-Brown MPV-950D module provides the analog to digital conversion facilities for the RTS [Burr-Brown, 1985]. It consists of

- a 12-bit A/D converter.
- 16 analog input channels,
- three analog input ranges: 0 to $+10V$. $\pm 5V$, or $\pm 10V$,
- 8 configurable input, operational amplifiers,
- and an external trigger input.

The 16 analog input channels feed into an analog multiplexer, which allows a single 12-bit A/D converter to convert each of them into a digital result. Eight of the channels feed through operational amplifiers that researchers can configure to accept differential analog inputs and provide various gains.

At 12 bits of resolution the module can sustain a sampling rate of 330,000 samples/sec. and at 8 bits of resolution it can sustain 500,000 samples/sec. The number of channels in use at a time determines what the sampling rate per channel will be.

The A/D module appears as 64 consecutive memory locations in the memory map of the SBC; each channel has a separate address. The SBC maintains control of the A/D module by accessing a control and status register. To select a particular input to be sampled requires the SBC to read from the module using the address of the next input channel to be converted. The SBC will receive the digital result of the previous conversion when it does this. If the SBC has enabled external triggering, the converter will wait for the trigger before starting the conversion, otherwise it begins conversion immediately after the previous digital result has been read. The SBC can either poll the status register, or wait for an interrupt, to determine when a conversion is complete.

At high sampling rates it is necessary for the SBC to poll the A/D converter to prevent the overheads of responding to a VMEbus interrupt from slowing

down the sampling rate. At lower sampling rates, such as in slower real-time control applications, using interrupts allows the SBC to spend most of its time processing previous data.

Initially the A/D converter module would not function correctly when operating with the MVME-133. This was because the module designers had designed the module to operate with an earlier VMEbus specification when the CPU modules operated at much slower clock rates. After several modifications, the module now functions correctly. Thus even though asynchronous buses such as VMEbus should handle modules at different speeds, high speed modules may bring to the fore any timing bugs present in slower modules.

A.3.5 D/A Converter Module

The Burr-Brown MPV-954 module provides the digital to analog conversion facilities for the RTS [Burr-Brown, 1986]. It features

- eight 12-bit D/A output channels,
- four analog output ranges: 0 to $+5V$, 0 to $+10V$, $\pm 5V$, or $\pm 10V$,
- 4 kwords of on-board RAM,
- programmable conversion rate,
- continuous or one shot mode,
- an on-board timer and an external trigger to control conversion rates,
- and a reconstruction filter on each output.

The on-board RAM stores the yet to be converted digital data. The D/A converters access the data from RAM, and output it in analog form at a rate determined by a software configurable on-board timer or an external trigger.

The maximum rate at which the D/A converters can update their analog outputs is 857,000 samples/second. The D/A converters take $2\mu s$ to settle to within 0.1% of their final value after a full scale voltage step.

The start and the stop address registers define the section of data memory from which the D/A converters read the data. The SBC selects the number of channels required by an application by writing to the control register. To start the conversion process, the SBC can either read from, or write to, the start register.

The converters can run in either single shot or continuous mode. In single shot mode, the module outputs the data stored in data memory between the start and stop addresses, and stops after it outputs the data at the stop address. The D/A converters will hold the analog outputs corresponding to the last digital value they have converted. In continuous mode, the module continuously cycles through the data between the start and stop addresses. In this mode the D/A converter module is behaving like a signal generator with the waveform stored in digital form in the data memory. The data memory is dual ported, enabling the SBC to update the digital data while the D/A converters are updating their analog outputs. The SBC stops the conversion cycle by writing to the control register.

Both an internal trigger and an external trigger mode are available to control the conversion rate. If the module uses internal trigger mode, a rate timer controls the conversion rate. The rate timer is programmable in 500 $nsec$ increments from $1.5\mu s$ to $127.5\mu s$. Thus the slowest rate at which the converters will update their outputs is 980 samples/second under internal trigger mode. Alternatively an external trigger can start each conversion. This allows slower sampling rates, and allows the module to synchronize outputs with an external experiment. The external trigger can also act as an event trigger to start the conversions under the control of the rate timer. The

module provides an external trigger output to help synchronize conversions with an external experiment. This is particularly useful when the module uses internal triggering under the control of the rate timer.

The SBC can either program the D/A module to send an interrupt, or poll the status register, to determine when the module has completed a conversion cycle.

A.3.6 Timer/Parallel Interface Module

The RTS uses a locally designed and constructed timer/parallel interface (TPI) module to allow the RTS to send digital (TTL) control signals to external experiments and to monitor external digital signals. The TPI also provides a programmable timer as a response to a request by users of the RTS for a more flexible timer for generating interrupts at regular time intervals. The TPI features

- a single 8-bit timer/counter with 8 decade prescaler, which allows programmable time intervals from 1 microsecond to 2550 seconds,
- a single 16-bit parallel output port, with separately addressable upper and lower bytes,
- and a single 16-bit parallel input port, with separately addressable upper and lower bytes.

A.3.7 Data Link

The RTS uses a custom designed 16-bit wide parallel interface to provide a high speed data link to the VAX™ 11/780 general purpose computer system. The following describes some of the reasons behind building a custom link, and goes on to describe the data link.

Ethernet is a commonly used data link for the transfer of large blocks of information, such as program source files, between computers located in a local area such as a University campus [Metcalfe & Boggs, 1976]. The VAX 11/780 is part of a large network of campus computers that use Ethernet as their local area network. One option was to add the RTS to the existing Ethernet network. This would have inconvenienced other users of the Ethernet, by saturating the Ethernet network whenever the RTS transferred large amounts of data. Typical data rates for transferring files over the local network using the file transfer program *ftp* are about 30 kilobytes/sec. Another alternative was to create a separate Ethernet link between the VAX 11/780 and the RTS. This alternative would have required two new Ethernet interfaces and network software to run on the RTS, which made the option too expensive for the funds available for building the RTS. The remaining option was to build a simple low cost data link that would provide a transfer rate at least as fast as the VAX 11/780 could transfer information to disk, which is about 200 kilobytes/sec.

The custom designed data link is an extension of the DR11-C General Device Interface available for the VAX UNIBUS [Digital, 1976]. This interface provides the logic and buffer registers necessary for program controlled 16-bit wide parallel data transfers between the VAX and an external device. Figure A.4 shows the structure of the data link. The department already had a commercial interface board that provided the necessary interface logic for managing UNIBUS transfers. It does not take advantage of the VAX's direct memory access (DMA) facilities. The main use of the interface is to transfer data to and from files on the VAX's disk; DMA transfers will not overcome the disk transfer bottleneck. The additional design complexity of a DMA interface was not cost effective for the small increase in throughput it provides by off loading the data transfer task from the VAX's CPU.

The parallel interface for both the GPS and RTS consists of a 16-bit wide

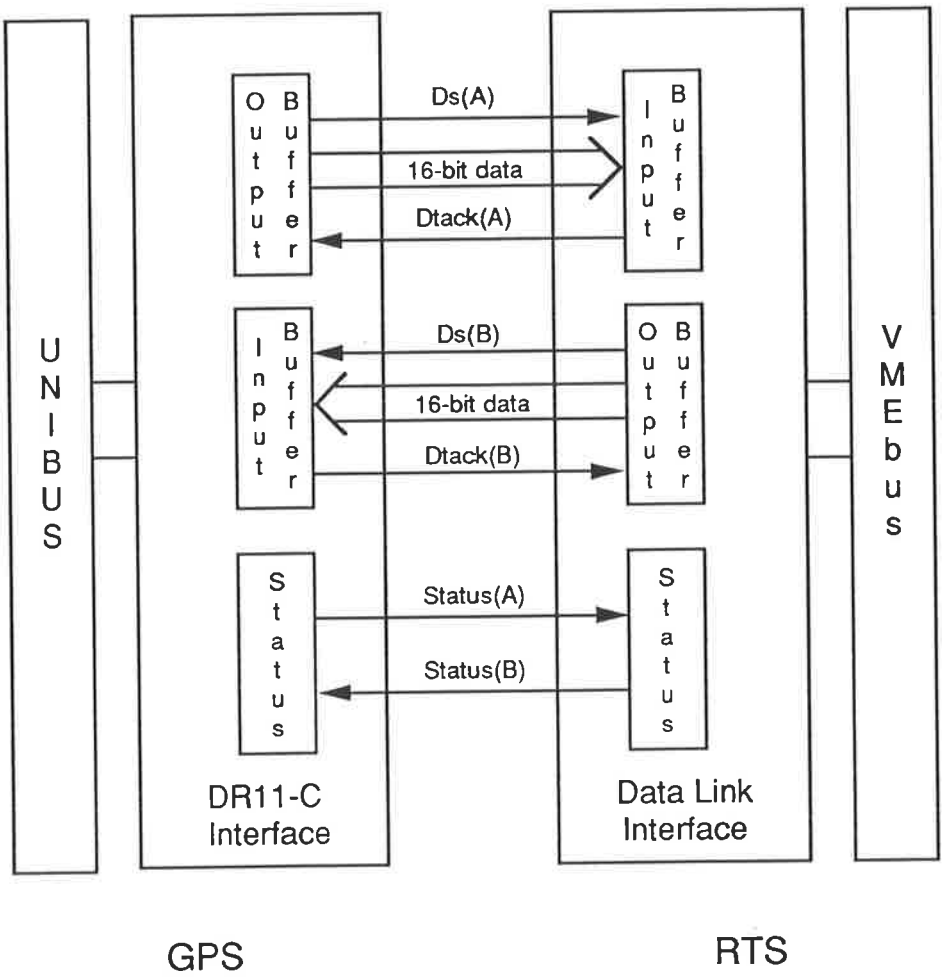


Figure A.4: Data Link structure

input buffer, a 16-bit wide output buffer, and a status register. The data link sends 16-bit wide words of data in parallel between the two interfaces, along with two status bits. The RTS controls one status bit, and the GPS controls the other. They provide a means of controlling the execution of the data transfer programs. When the RTS sends a 16-bit word of data, a control signal notifies the GPS that there is fresh data. Another control signal interrupts the RTS when the GPS has read the data. The RTS can then write another word of data into the output buffer. The transfer protocol is asynchronous; the transmitter does not send the next word of data until the receiver acknowledges receipt of the previous word of data. There is no hardware error checking; the data transfer software is responsible for checking for data errors.

A distance of about 100m separates the RTS from the GPS. To minimize the effect of electrical noise over this distance, the data link uses differential signals over standard twisted pair telephone cable. The data link cabling is cheap, and has a measured delay of about 100ns. The parallel interfaces deskew the parallel data lines before storing data in buffers. As researchers use the RTS in an electrically clean room, the data link includes optical isolation at the RTS end of the link.

The data transfer rate between the GPS's disk storage and the RTS is currently 50 kilobytes/sec. This is fast enough to transfer programs and data to the RTS. The data rate can probably increase by a factor of 4 by allowing the data transfer software to poll the parallel interfaces to determine when they need service, thereby removing interrupt latencies. This would bring the data transfer rate up to the limit of the disk transfer rate, and would be useful for applications that need to acquire several megabytes of data in real time.

A.3.8 Software Support

The Department purchased the Amsterdam Compiler Kit to provide cross compiler facilities on the GPS for generating MC68020 machine code for the RTS [Tanenbaum et al., 1983]. Users of the RTS mainly use the C programming language. A set of C library routines is available for simplifying the programmer's interface to the various facilities available on the RTS. There are routines for configuring the RTS for a particular experiment, and simple routines for controlling the input and output of information during an experiment. The compiler has locally added extensions for using the floating point capabilities of the MC68881.

A.3.9 Summary

The RTS is now in use by researchers experimenting with techniques for identifying the parameters of an industrial system to allow the design of an efficient system controller. The RTS induces changes in the operating point of an external system and observes the system's response. Researchers use the mathematical software available on the GPS to analyse experimental data from the RTS and determine the parameters of the system. Researchers will eventually use the RTS for evaluating algorithms, based on experimental parameters, for controlling a system.

The design has met the original specifications at a cost of about A$ 20,000. Most of the cost came from the purchase of the VMEbus backplane with associated power supply and cooling, the single board computer, and the analog I/O boards. These analog I/O boards represented the best compromise between flexibility and cost. They have ample channels, and have adjustable conversion rates to achieve different levels of resolution. Conversion rates exceeding 500 kilosamples/sec are possible. The single board computer was

among the highest performance boards available at the time of purchase. The simple, low cost data link to the GPS kept costs down by allowing the GPS to provide software development facilities, and off-line analysis of results for the RTS.

Now that smaller portable workstations can achieve similar performance as the VAX 11/780, using a workstation as the general purpose computer will make the RTS portable. This is useful for researchers wanting to test control algorithms outside the laboratory environment.

A.3.10 Acknowledgements

I wish to thank Norman Blockley and John Hunt for their work in constructing circuits, Alf Grasso for his work on the software library for the RTS. Special thanks to Michael Liebelt for providing general encouragement and support for the work, and for developing the TPI module. Recognition must also go to the undergraduate students who have worked with the early versions of the RTS: J. Schutz, Richard Middelmann and Andrew Beaumont-Smith.

Bibliography

Ackland. B. D. and Clark, R. A., 1989. Event-EMU: An Event Driven Timing Simulator for MOS VLSI circuits. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 80–83, IEEE Computer Society. November.

Ackland. B. D., 1988. Knowledge-Based Physical Design Automation. In Preas, B. T. and Lorenzetti, M. J., editors, *Physical Design Automation of VLSI Systems*, chapter 9, Benjamin/Cummings, Menlo Park, CA, USA.

Ackland. B. D., Ahuja, S. R., Lindstrom, T. L., and Romero, D. J., 1985. CEMU - A Concurrent Timing Simulator. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 122–124, IEEE Computer Society. November.

Ackland. B., Ahuja, S.. DeBenedictis, E., London, T., Lucco, S., and Romero. D.. 1986. MOS Timing Simulation on a Message Based Multiprocessor. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 446–450, IEEE Computer Society, October.

Agarwal. A., Simoni, R., Hennessy, J., and Horowitz, M., 1988. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA)*, pages 280–289, ACM SIGARCH, Computer Architecture News 16(2), May.

Agrawal. P. and Dally, W. J., 1990. A Hardware Logic Simulation System. *IEEE Transactions on Computer-Aided Design*, 9(1):19–29, January.

Agrawal. R. and Jagadish, H. V., 1988. Partitioning Techniques for Large-Grained Parallelism. *IEEE Transactions on Computers*, 37(12):1627–1634, December.

Agrawal. P.. 1986. Concurrency and Communication in Hardware Simulators. *IEEE Transactions on Computer-aided Design*, CAD-5(4):617–623, October.

Agrawal, P., Dally, W. J., Fischer, W. C., Jagadish, H. V., Krishnakumar, A. S., and Tutundjian, R., 1987. MARS: A Multiprocessor-based Programmable Accelerator. *IEEE Design & Test of Computers*, 4(5):28–37, October.

Ahuja, S. R., 1983. S/NET: A High-Speed Interconnect for Multiple Computers. *IEEE Journal on Selected Area in Communications*, SAC-1(5):751–756, November.

Ahuja, S., Carriero, N., and Gelernter, D., 1986. Linda and Friends. *Computer*, 19(8):26–34, August.

Ahuja, S., Carriero, N. J., and Krishnaswamy, D. H. G. V., 1988. Matching Language and Hardware for Parallel Computation in the Linda Machine. *IEEE Transactions on Computers*, 37(8):921–929, August.

Amdahl, G. M., 1967. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30:483–485, spring.

Annaratone, M., Pommerell, C., and Rühl, R., 1989. Interprocessor Communications Speed and Performance in Distributed-memory Parallel Processors. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*, pages 315–324, ACM SIGARCH, Computer Architecture News 17(3), June.

Armstrong, D. B., 1972. A Deductive Method for Simulating Faults in Logic Circuits. *IEEE Transactions on Computers*, C-21(5):464–471, May.

Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D., and Steenkiste, P. A., 1989. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 205–216, ACM, Computer Architecture News 17(2), April.

Asano, T., Sato, M., and Ohtsuki, T., 1986. Computational Geometry Algorithms. In Ohtsuki, T., editor, *Layout Design and Verification*, chapter 9, Elsevier Science, Amsterdam, The Netherlands.

Ashenden, P. J. and Marlin, C. D., 1988. A Behavioural Specification of Cache Coherence. *The Australian Computer Journal*, 20(2):50–57, May.

Ashenden, P. J., Barter, C. J., and Marlin, C. D., 1987. The Leopard Workstation Project. *Computer Architecture News*, 15(4):40–50, September.

Ashenden, P. J., Barter, C. J., and Petty, M. A., 1989. *The Leopard Multiprocessor Workstation Project.* Technical Report LW-1, University of Adelaide, Centre for Computer Systems and Software Engineering, November.

Ashok, V., Costello, R., and Sadayappan, P., 1985. Distributed Discrete Event Simulation using Dataflow. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 503–510, IEEE Computer Society, August.

Ashok, V., Costello, R., and Sadayappan, P., 1985. Modeling Switch-Level Simulation using Dataflow. In *Proceedings of the 22nd Design Automation Conference*, pages 637–644, ACM/IEEE, June.

Athas, W. C. and Seitz, C. L., 1988. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 21(8):9–24, August.

August, M. C., Brost, G. M., Hsiung, C. C., and Schiffleger, A. J., 1989. Cray X-MP: The Birth of a Supercomputer. *Computer*, 22(1):45–52, January.

Bailey, M. L. and Snyder, L., 1989. A Model for Comparing Synchronization Strategies for Parallel Logic-Level Simulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 502–505, IEEE Computer Society, November.

Bal, H. E. and Tanenbaum, A. S., 1988. Distributed Programming with Shared Data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91, IEEE Computer Society, October.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., 1989. A Distributed Implementation of the Shared Data-Object Model. In *Proceedings of the 1st USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, USENIX, October.

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., 1989. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., 1990. Experience with Distributed Programming in Orca. In *Proceedings of the International Conference on Computer Languages*, pages 79–89, IEEE Computer Society, March.

Banerjee, P., 1988. Parallel Algorithms for Design Verification. In *A Tutorial on the Use of Parallel Processing in VLSI CAD Applications*, IEEE Computer Society, International Conference on Computer Aided Design (ICCAD), November.

Banerjee, P., Jones, M. H., and Sargent, J. S., 1990. Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):91–106, January.

Batcher, K. E., 1974. STARAN parallel processor system hardware. *Proceedings of the National Computer Conference (AFIPS)*, 43:405–410.

Batcher, K. E., 1980. Design of a Massively Parallel Processor. *IEEE Transaction on Computers*, C-29(9):836–840, September.

Bays, L. E., Chen, C., Fields, E. M., Gadenz, R. N., Hays, W. P., Moscovitz, H. S., and Szymanski, T. G., 1989. Post-layout Verification of the WE DSP32 Digital Signal Processor. *IEEE Design and Test of Computers*, 6(1):56–65, February.

Belkhale, K. P. and Banerjee, P., 1988. PACE: A Parallel VLSI Extractor on the Intel Hypercube. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 326–329, IEEE Computer Society, November.

Belkhale, K. P. and Banerjee, P., 1989. PACE2: An Improved Parallel VLSI Extractor with Parameter Extraction. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 526–529, IEEE Computer Society, November.

Bell, C. G., 1985. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–467, 26 April.

Bell, G., 1989. The Future of High Performance Computers in Science and Engineering. *Communications of the ACM*, 32(9):1091–1101, September.

Bentley, J. L., Haken, D., and Hon, R. W., 1980. *Statistics on VLSI Designs.* Technical Report CMU-CS-80-111, Carnegie-Mellon University, Pittsburgh, Penn., USA, April.

Berra, P. B., Ghafoor, A., Guizani, M., Marcinkowski, S. J., and Mitkas, P. A., 1989. Optics and Supercomputing. *Proceedings of the IEEE*, 77(12):1797–1815, December.

Bier, G. E. and Pleszkun, A. R., 1985. An Algorithm for Design Rule Checking on a Multiprocessor. In *Proceedings of the 22nd Design Automation Conference*, ACM/IEEE, June.

Birrell, A. D. and Nelson, B. J., 1984. Implementing Remote Procedure Calls. *ACM Transactions on Computing Systems*, 2(1):39–59, February.

Black. D. L.. 1990. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer*, 23(5):35–43, May.

Blackett, R. K., 1983. UNIX-based system runs real-time applications. *Mini-micro Systems*, May.

Blair. G. S., Malone, J. R., and Mariani, J. A., 1985. A Critique of UNIX. *Software' - Practice and Experience*, 15(12):1125–1139, December.

Blank. T., 1984. A Survey of Hardware Accelerators Used in Computer-Aided Design. *IEEE Design & Test*, 1:21–39, August.

Blank. T., 1990. The MasPar MP-1 Architecture. In *Proceedings of the COMPCON Spring Conference*, pages 20–24, IEEE Computer Society, February.

Blank, T., Stefik, M., and vanCleemput, W., 1981. A Parallel Bit Map Processor Architecture for DA Algorithms. In *Proceedings of the 18th Design Automation Conference*, ACM/IEEE, June.

Borrill, P. L., 1985. MicroStandards Special Feature: A Comparison of 32-Bit Buses. *IEEE Micro*, :71–79, December.

Borrill, P. L.. 1986. Objective comparison of 32-bit buses. *Microprocessors and Microsystems*, 10(2):94–100, March.

Borrill, P. L., 1989. High-speed 32-bit buses for forward-looking computers. *IEEE Spectrum*, :34–37, July.

Bosshart, P., 1987. Overview of a 32-bit Microprocessor Design Project. In Fichtner, W. and Morf, M., editors, *VLSI CAD Tools and Applications*, chapter 12, Kluwer, Norwell, Mass., USA.

Bottorff, P. S., 1981. Computer Aids to Testing - An Overview. In Antognetti, P., Pederson, D. O., and De Man, H., editors, *Computer Design Aids for VLSI Circuits*, pages 417–463, Sijthoff & Noordhoff, Alphen aan den Rijn, The Netherlands.

Boxer, A., 1983. Designing for High Performance Data Acquisition. *Computer Design*, September.

Brady, J. T., 1986. A Theory of Productivity in the Creative Process. *IEEE Computer Graphics and Applications (CG&A)*, :25–34, May.

Breuer, M. A. and Friedman, A. D., 1976. *Diagnosis & Reliable Design of Digital Systems.* Computer Science Press, Woodland Hills, CA, USA.

Breuer, M. A., 1972. *Design Automation of Digital Systems. Volume 1: Theory and Techniques.* Prentice-Hall, Englewood Cliffs, NJ, USA.

Briner. Jr., J. V., Ellis, J. L., and Kedem, G., 1988. Taking Advantage of Optimal On-Chip Parallelism for Parallel Discrete Event Simulation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 312–315, IEEE Computer Society, November.

Broomell, G. and Heath, J. R., 1983. Classification Categories and Historical Development of Circuit Switching Topologies. *ACM Computing Surveys*, 15(2):95–133, June.

Brouwer, R. and Banerjee, P., 1988. A Parallel Simulated Annealing Algorithm for Channel Routing on a Hypercube Multiprocessor. In *Proceedings of the International Conference on Computer Design (ICCD)*, page , IEEE Computer Society, October.

Brouwer, R. J. and Banerjee, P., 1990. PHIGURE: A Parallel Hierarchical Global Router. In *Proceedings of the 27th Design Automation Conference (DAC)*. pages 650–653, ACM/IEEE, June.

Bryant, R. E., 1984. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, C-33(2):160–177. February.

Bryant. R. E., 1988. Data Parallel Switch-Level Simulation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 354–357, IEEE Computer Society, November.

Burr-Brown., 1985. *MPV950D High Speed Analog Input Board Operating Manual.* Burr-Brown Limited, DSP Group, Livingston, Scotland, e edition.

Burr-Brown., 1986. *MPV954 High Speed Analog Output Board Operating Manual.* Burr-Brown Limited, DSP Systems Division, Livingston, Scotland, 2.b edition.

Cajani. V., 1987. Architectural Issues in Designing a UNIX Multiprocessor System. *Microprocessing and Microprogramming*, 20:79–84.

Cane, D. and Mullen, S., 1984. Triple-bus Architecture gains speed, versatility. *Computer Design*, February.

Carlson, E. C. and Rutenbar, R. A., 1987. A Scanline Data Structure Processor for VLSI Geometry Checking. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):780–794, September.

Carlson, E. C. and Rutenbar, R. A., 1988. Mask Verification on the Connection Machine. In *Proceedings of the 25th Design Automation Conference*, pages 134–140, ACM/IEEE. June.

Carlson, E. C. and Rutenbar, R. A., 1990. Design and Performance Evaluation of New Massively Parallel VLSI Mask Verification Algorithms in JIGSAW. In *Proceedings of the 27th Design Automation Conference (DAC)*, pages 253–259, ACM/IEEE, June.

Carriero, N. and Gelernter, D., 1989. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September.

Carriero, N. and Gelernter, D., 1989. Linda in Context. *Communications of the ACM*, 32(4):444–458, April.

Casotto, A. and Sangiovanni-Vincentelli, A., 1987. Placement of Standard Cells Using Simulated Annealing on the Connection Machine. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 350–353, IEEE Computer Society, November.

Casotto, A., Romeo, F., and Sangiovanni-Vincentelli, A., 1987. A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):838–847, September.

Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A., 1990. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer*, 23(6):49–58, June.

Chamberlain, R. D. and Franklin, M. A., 1988. Discrete-Event Simulation on Hypercube Architectures. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 272–275, IEEE Computer Society, November.

Chandra, S. and Patel, J. H., 1988. Test Generation in a Parallel Processing Environment. In *Proceedings of the International Conference on Computer Design (ICCD)*, page , IEEE Computer Society, October.

Chandy, K. M. and Misra, J., 1981. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, April.

Chang, M. and Hajj, I. N., 1988. iPRIDE: A Parallel Integrated Circuit Simulator Using Direct Method. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 304–307, IEEE Computer Society, November.

Chawla, B. R., Gummel, H. K., and Kozak, P., 1975. MOTIS - An MOS Timing Simulator. *IEEE Transactions on Circuits and Systems*, CAS-22(12):901–910, December.

Chiang, K-W., Nahar, S., and Lo, C-Y., 1989. Time Efficient VLSI Artwork Analysis Algorithms in GOALIE2. *IEEE Transactions on Computer-Aided Design*. 8(6):640–648, June.

Chlamtac, I. and Franta, W. R., 1990. Rationale, Directions, and Issues Surrounding High Speed Networks. *Proceedings of the IEEE*, 78(1):94–120, January.

Christy, P., 1990. Software To Support Massively Parallel Computing on the MasPar MP-1. In *Proceedings of the COMPCON Spring Conference*, pages 29–33, IEEE Computer Society, February.

Chua, L. O. and Lin, P., 1975. *Computer-aided Analysis of Electronic Circuits*. Prentice-Hall, Englewood Cliffs, NJ, USA.

Chung, M. J. and Chung, Y., 1989. Data Parallel Simulation using Time-Warp on the Connection Machine. In *Proceedings of the 26th Design Automation Conference (DAC)*, pages 98–102, ACM/IEEE, June.

Cohn, R., Gross, T., Lam, M., and Tseng, P., 1989. Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 2–14, ACM, Computer Architecture News 17(2), April.

Cox, P., Burch, R., and Epler, B., 1986. Circuit Partitioning for Parallel Processing. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 186–189, IEEE Computer Society, November.

Crowther, W., Goodhue, J., Starr, E., Thomas, R., Milliken, W., and Blackadar, T., 1985. Performance Measurements on a 128-node Butterfly Parallel Processor. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 531–540, IEEE Computer Society, August.

Dally, W. J., 1987. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers.

Darema, F. and Pfister, G. F., 1987. Multipurpose Parallelism for VLSI CAD on the RP3. *IEEE Design & Test of Computers*, 4(5):19–27, October.

De Man, H., Arnout, G., and Reynaert, P., 1981. Mixed-mode Circuit Simulation Techniques and Their Implementation in DIANA. In Antognetti, P., Pederson, D. O., and De Man, H., editors, *Computer Design Aids for VLSI Circuits*, pages 113–174, Sijthoff & Noordhoff, Alphen aan den Rijn, The Netherlands.

Denning, P. J., 1970. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, September.

Dennis, J. B., 1979. The Varieties of Data Flow Computers. In *Proceedings of the 1st International Conference on Distributed Computing Systems*, pages 430–439, IEEE Computer Society, October.

Deutsch, J. T. and Newton, A. R., 1984. A Multiprocessor Implementation of Relaxation-Based Electrical Circuit Simulation. In *Proceedings of the 21st Design Automation Conference (DAC)*, pages 350–357, ACM/IEEE, June.

Deutsch, J. T., Lovett, T. D., and Squires, M. L., 1986. Parallel Computing for VLSI Circuit Simulation. *VLSI Systems Design*, :46–52, July.

Dewey Jr., C. F., 1983. Multiprocessor system shifts data acquisition into overdrive. *Industrial Research & Development*, May. . .

Dibble, P. C. and Scott, M. L., 1989. Beyond Striping: The Bridge Multiprocessor File System. *Computer Architecture News*, 17(5):32–39, September.

Dibble, P. C., Scott, M. L., and Ellis, C. S., 1988. Bridge: A High-Performance File System for Parallel Processors. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 154–161, IEEE Computer Society, June.

Digital., 1976. General Device Interface, DR11-C. In *PDP-11 Peripherals Handbook*, pages 4-217–4-226, Digital Equipment Corporation, Maynard, Massachusetts, USA.

Dinning, A., 1989. A Survey of Synchronization Mehtods for Parallel Computers. *Computer*, 22(7):66–77, July.

Ditzel, D. R. and Berenbaum, A., 1987. Experience with CAD Tools for a 32-Bit VLSI Microprocessor. In Fichtner, W. and Morf, M., editors, *VLSI CAD Tools and Applications*, chapter 11, Kluwer, Norwell, Mass., USA.

Dongarra, J. J., editor. *Experimental Parallel Computing Architectures.* Volume 1 of *Special Topics in Supercomputing*, Elsevier Science, Amsterdam, The Netherlands.

D&T., 1989. A D&T Roundtable: Expert Systems in CAD. *IEEE Design and Test of Computers*, 6(4):61–68, August.

D&T., 1989. A D&T Roundtable: Mixed-Mode Simulation. *IEEE Design and Test of Computers*, 6(1):67–75, February.

Dumlugöl, D., Odent, P., Cockx, J. P., and Man, H. J. D., 1987. Switch-Electrical Segmented Waveform Relaxation for Digital MOS VLSI and Its Acceleration on Parallel Computers. *IEEE Transactions on Computer Aided Design*, CAD-6(6):992–1005, November.

Duncan, R., 1990. A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16, February.

Durand, M. D., 1989. Parallel Simulated Annealing: Accuracy vs. Speed in Placement. *IEEE Design & Test of Computers*, 6(3):8–34, June.

Eager, D. L., Zahorjan, J., and Lazowska, E. D., 1989. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, March.

Feng, T., 1981. A Survey of Interconnection Networks. *Computer*, :12–27, December.

Fleisch, B. D. and Popek, G. J., 1989. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 211–223, ACM SIGOPS, Operating Systems Review 23(5), December.

Flynn, M. J., 1966. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, December.

Flynn, M. J., 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September.

Fortes, J. A. B. and Wah, B. W., 1987. Systolic Arrays - From Concept to Implementation. *Computer*, 20(7):12–17, July.

Foulser, D. E. and Schreiber, R., 1987. The Saxpy Matrix-1: A General-Purpose Systolic Computer. *Computer*, 20(7):35–43, July.

Frank, E. H., 1986. Exploiting Parallelism in a Switch-Level Simulation Machine. In *Proceedings of the 23rd Design Automation Conference (DAC)*, pages 20–26, June.

Franklin, M. A., Wann, D. F., and Wong, K. F., 1984. Parallel Machines and Algorithms for Discrete-Event Simulation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 449–458, IEEE Computer Society, August.

Fujiwara, H. and Inoue, T., 1989. Optimal Granularity of Test Generation in a Distributed System. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 158–161, IEEE Computer Society, November.

Fujiwara, H. and Motohara, A., 1988. Fast Test Pattern Generation Using a Multiprocessor System. *Transactions of the Institute of Electronics, Information, and Communication Engineers (IEICE) (Japan)*, E-71(4):441–447, April.

Fujiwara, H. and Shimono, T., 1983. On the Acceleration of Test Generation Algorithms. *IEEE Transactions on Computers*, C-32(12):1137–1144, December.

Gabber, E., 1990. VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):304–317, July.

Gaglianello, R. D. and Katseff, H. P., 1985. Meglos: An Operating System for a Multiprocessor Environment. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 35–42, IEEE Computer Society, May.

Gaglianello, R. D. and Katseff, H. P., 1986. Communications in Meglos. *Software - Practice and Experience*, 16(10):945–963, October.

Gaglianello, R. D., Robinson, B. S., Lindstrom, T. L., and Sampieri, E. E., 1989. HPC/VORX: A Local Area Multicomputer System. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, IEEE Computer Society, June.

Gai, S., Montessoro, P. L., and Somenzi, F., 1988. MOZART: A Concurrent Multilevel Simulator. *IEEE Transactions on Computer-Aided Design*, 7(9):1005–1016, September.

Gajski, D. D. and Lin, Y. S., 1988. Module Generation and Silicon Compilation. In Preas, B. T. and Lorenzetti, M. J., editors, *Physical Design Automation of VLSI Systems*, chapter 7, Benjamin/Cummings, Menlo Park, CA, USA.

Garey, M. R. and Johnson, D. S., 1979. *Computers and Intractability.* W. H. Freeman, San Francisco.

Gehringer, E. F., Abullarade, J., and Gulyn, M. H., 1988. A Survey of Commercial Parallel Processors. *Computer Architecture News*, 16(4):75–107, September.

Gerner, M. and Johansson, M., 1987. VLSI Testing: DFT Strategies and CAD Tools. In Fichtner, W. and Morf, M., editors, *VLSI CAD Tools and Applications*, chapter 19, Kluwer, Norwell, Mass., USA.

G. F. Pfister, W. C. B., George, D. A., Harvey, S. L., Klienfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., and Weiss, J., 1987. An Introduction to the IBM Research Parallel Processor Prototype (RP3). In Dongarra, J. J., editor, *Experimental Parallel Computing Architectures*, pages 123–140, Elsevier Science, Amsterdam, The Netherlands.

Gibson, G. A., Hellerstein, L., Karp, R. M., Katz, R. H., and Patterson, D. A., 1989. Failure Correction Techniques for Large Disk Arrays. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 123–132, ACM, Computer Architecture News 17(2), April.

Giloi, W. K., 1988. SUPRENUM: A trendsetter in modern supercomputer development. *Parallel Computing*, 7:283–296.

Gimarc, C. E. and Milutinovic, V. M., 1987. A Survey of RISC Processors and Computers of the Mid-1980's. *Computer*, 20(9):59–69, September.

Gingell, R. A., Moran, J. P., and Shannon, W. A., 1987. Virtual Memory Architecture in SunOS. In *Summer Conference Proceedings, Phoenix*, pages 81–94, USENIX Association.

Gregoretti, F. and Segall, Z., 1984. Analysis and Evaluation of VLSI Design Rule Checking implementation in a multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 7–14, IEEE Computer Society, August.

Gregoretti, F. and Segall, Z., 1987. Analysis and Evaluation of Parallel Rectangle Intersection for VLSI Design Rule Checking. *Microprocessing and Microprogramming*, 19(2):85–100, February.

Harrison, D. S., Newton, A. R., Spickelmier, R. L., and Barnes, T. J., 1990. Electronic CAD Frameworks. *Proceedings of the IEEE*, 78(2):393–417, February.

Hawley, D., 1989. Superfast Bus supports Sophisticated Transactions. *High Performance Systems*, :90–94, September.

Hayes, J. P. and Mudge, T., 1989. Hypercube Supercomputers. *Proceedings of the IEEE*, 77(12):1829–1841, December.

Hayes, J. P., Mudge, T. N., Stout, Q. F., Colley, S., and Palmer, J., 1986. Architecture of a Hypercube Supercomputer. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 653–660, IEEE Computer Society.

Haynes, L. S., Lau, R. L., Siewiorek, D. P., and Mizell, D. W., 1982. A Survey of Highly Parallel Computing. *Computer*, :9–24, January.

Helin, J. and Kaski, K., 1989. *Performance Analysis of High-speed Computers II.* Technical Report Electronics Laboratory Report 3-89, Tampere University of Technology, Finland, June.

Hennessy, J. L. and Patterson, D. A., 1990. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, USA.

Hill et al., M., 1986. Design Decisions in SPUR. *Computer*, 19(11):8–22, November.

Hillis, W. D. and Steele, Jr., G. L., 1986. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December.

Hillis, W. D., 1982. New Computer Architectures and Their Relationship to Physics or Why Computer Science Is No Good. *International Journal of Theoretical Physics*, 21(3/4):255–262.

Holden, T., 1987. *Knowledge Based CAD and Micro-electronics.* Elsevier Science, Amsterdam, The Netherlands.

Hon, R. W., 1987. Dynamic Analysis Tools. In Rubin, S. M., editor, *Computer Aids for VLSI Design*, chapter 6, Addison-Wesley, Reading, Mass., USA.

Howe, C. D. and Moxon, B., 1987. How to program parallel processors. *IEEE Spectrum*, :36–41, September.

Howe, C. D. and Moxon, B., 1987. How to program parallel processors. *IEEE Spectrum*, 24(9):36–41, September.

Hung, G. G., Wen, Y. C., Gallivan, K., and Saleh, R., 1990. Parallel Circuit Simulation Using Hierarchical Relaxation. In *Proceedings of the 27th Design Automation Conference (DAC)*, pages 394–399, ACM/IEEE, June.

Hwang, K., 1987. Advanced Parallel Processing with Supercomputer Architectures. *Proceedings of the IEEE*, 75(10):1348–1379, October.

IEEE P1596., 1990. *Part 1: An Overview of the Scalable Coherent Interface.* IEEE Computer Society, draft 0.59 edition, February.

IEEE P896.1., 1990. *Futurebus +.* IEEE Computer Society, Washington, DC, draft 8.2 edition, February.

Itai, A. and Raz, Y., 1988. The Number of Buffers Required for Sequential Processing of a Disk File. *Communications of the ACM*, 31(11):1338–1342, November.

Jacob, G. K., Newton, A. R., and Pederson, D. O., 1986. An Empirical Analysis of the Performance of a Multiprocessor-Based Circuit Simulator. In *Proceedings of the 23rd Design Automation Conference (DAC)*, pages 588–593, ACM/IEEE, June.

Jain, S. K. and Agrawal, V. D., 1985. Modeling and Test Generation Algorithms for MOS Circuits. *IEEE Transactions on Computers*, C-34(5):426–433, May.

James, D. V., Laundrie, A. T., Gjessing, S., and Sohi, G. S., 1990. Distributed-Directory Scheme: Scalable Coherent Interface. *Computer*, 23(6):74–77, June.

Jayaraman, R. and Rutenbar, R., 1987. Floorplanning by Annealing on a Hypercube Multiprocessor. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 346–349, IEEE Computer Society, November.

Jefferson, D. R., 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July.

Jesshope, C. R., Miller, P. R., and Yantchev, J. T., 1989. High Performance Communications in Processor Networks. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*, pages 150–157, ACM SIGARCH, Computer Architecture News 17(3), June.

Johnson, E. E., 1988. Completing an MIMD Multiprocessor Taxonomy. *Computer Architecture News*, 16:44–47, July.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C., 1987. *Optimization by Simulated Annealing: An Experimental Evaluation (Part 1)*. Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, USA.

Jones, M. and Banerjee, P., 1987. Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube. In *Proceedings of the 24th Design Automation Conference (DAC)*, pages 807–813, ACM/IEEE, June.

Jones, T., 1989. Engineering Design of the Convex C2. *Computer*, 22(11):36–44, January.

Jouppi, N. P. and Wall, D. W., 1989. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 272–282, ACM, Computer Architecture News 17(2), April.

Jouppi, N. P., 1987. Timing Analysis and Performance Improvement of MOS VLSI Designs. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):650–665, July.

Jouppi, N. P., Bertoni, J., and Wall, D. W., 1989. A Unified Vector/Scalar Floating-Point Architecture. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 134–143, ACM, Computer Architecture News 17(2), April.

Kane, R. and Sahni, S., 1987. A Systolic Design-Rule Checker. *IEEE Transactions on Computer-aided Design*, CAD-6(1):22–32, January.

Karp, A. H., 1987. Programming for Parallelism. *IEEE Computer*, 20(5):43–57, May.

Katseff, H. P., 1987. Flow-Controlled Multicast in Multiprocessor Systems. In *Proceedings of the 6th Phoenix Conference on Computers and Communication*, pages 8–13, IEEE Computer Society, February.

Katz, R. H., Gibson, G. A., and Patterson, D. A., 1989. Disk System Architectures for High Performance Computing. *Proceedings of the IEEE*, 77(12):1842–1858, December.

Katz, R. H., Ousterhout, J. K., Patterson, D. A., Chen, P., Chervenak, A., Drewes, R., Gibson, G., Lee, E., Lutz, K., Miller, E., and Rosenblum, M., 1989. A Project on High Performance I/O Subsystems. *Computer Architecture News*, 17(5):24–31, September.

Kim, M. Y., 1985. Parallel Operation of Magnetic Disk Storage Devices: Synchronized Disk Interleaving. In *Proceedings of the 4th International Workshop on Database Machines*, pages 300–329, March.

Kim, M. Y., 1986. Synchronized Disk Interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November.

Kirkland, T. and Mercer, M. R., 1988. Algorithms for Automatic Test Pattern Generation. *IEEE Design and Test of Computers*, 5(3):43–55, June.

Kirkpatrick, S., Gelatt, Jr., C. D., and Vecchi, M. P., 1983. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 13 May.

Kling, R. M. and Banerjee, P., 1989. ESP: Placement by Simulated Evolution. *IEEE Transactions on Computer-aided Design*, 8(3):245–256, March.

Kling, R. and Banerjee, P., 1990. Optimization by Simulated Evolution with Applications to Standard Cell Placement. In *Proceedings of the 27th Design Automation Conference (DAC)*, pages 20–25, ACM/IEEE, June.

Kohn, L. and Margulis, N., 1989. Introducing the Intel i860 64-Bit Microprocessor. *IEEE Micro*, 9(4):15–30, August.

Kramer, G. A., 1983. Employing Massive Parallelism in Digital ATPG Algorithms. In *Proceedings of the International Test Conference*, pages 108–114, IEEE Computer Society, October.

Kravitz, S. A. and Rutenbar, R. A., 1987. Placement by Simulated Annealing on a Multiprocessor. *IEEE Transactions on Computer-aided Design*, CAD-6(4):534–549, July.

Kravitz, S. A., Bryant, R. E., and Rutenbar, R. A., 1989. Logic Simulation on Massively Parallel Architectures. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*, pages 336–343, ACM SIGARCH, Computer Architecture News 17(3), June.

Kravitz, S. A., Bryant, R. E., and Rutenbar, R. A., 1989. Massively Parallel Switch-Level Simulation: A Feasibility Study. In *Proceedings of the 26th Design Automation Conference (DAC)*, pages 91–97, ACM/IEEE, June.

Krishnaswamy, V., Ahuja, S., Carriero, N., and Gelernter, D., 1988. The Architecture of a Linda Coprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA)*, pages 240–249, ACM SIGARCH, Computer Architecture News 16(2), May.

Kronenberg, N. P., Levy, H. M., and Strecker, W. D., 1986. VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2):130–146, May.

Kuck, D. J., 1982. High-speed machines and their compilers. In Evans, D. J., editor, *Parallel Processing Systems*, chapter 11, Cambridge University Press. Cambridge, UK.

Kung, H. T., 1982. Why Systolic Architectures? *Computer*, 15(1):37–46, January.

Kung, H. T., 1986. Memory Requirements for Balanced Computer Architectures. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 49–54, ACM SIGARCH, June.

Kung, S.. Lo, S., Jean, S., and Hwang, J., 1987. Wavefront Array Processors - Concept to Implementation. *Computer*, 20(1):18–33, July.

Lauther, U., 1981. An O(NlogN) Algorithm for Boolean Mask Operations. In *Proceedings of the 18th Design Automation Conference*, pages 555–562, ACM/IEEE, June.

Lelarasmee, E., Ruheli, A., and Sangiovanni-Vincentelli, A. L., 1982. The waveform relaxation method for the time-domain analysis of large scale integrated circuits. *IEEE Transactions on Computer-Aided Design of ICAS*, CAD-1(3):131–145, August.

Leler, W., 1990. Linda Meets Unix. *Computer*, 23(2):43–54, February.

Lengauer, T., 1988. The Combinatorial Complexity of Layout Problems. In Preas, B. T. and Lorenzetti, M. J., editors, *Physical Design Automation of VLSI Systems*, chapter 10, Benjamin/Cummings, Menlo Park, CA, USA.

Levendel, Y. H., Menon, P. R., and Patel, S. H., 1983. Parallel Fault Simulation Using Distributed Processing. *The Bell System Technical Journal*, 62(10):3107–3137, December.

Levitin, S. M., 1986. *MACE: A Multiprocessor Approach to Circuit Extraction.* Master's thesis, Massachusetts Institute of Technology (MIT), Cambridge, Mass., USA, June.

Levreault Jr., J. E., 1983. High-speed data acquisition design system. *Electronic Products Magazine*, :89–92, 4 March.

Lewis, D. M., 1988. Hardware Accelerators for Timing Simulation of VLSI Digital Circuits. *IEEE Transactions on Computer-Aided Design*, 7(11):1134–1149, November.

Li, K. and Hudak, P., 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November.

Li, K., 1986. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, New Haven, CT, USA.

Lorenzetti, M. J. and Baeder, D. S., 1988. Routing. In Preas, B. T. and Lorenzetti, M. J., editors, *Physical Design Automation of VLSI Systems*, chapter 5, Benjamin/Cummings, Menlo Park, CA, USA.

Macomber, S. and Mott, G., 1985. Hardware Acceleration for Layout Verification. *VLSI Design*, :18–27, July.

Marantz, J. D., 1986. Exploiting Parallelism In VLSI CAD. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 442–445, IEEE Computer Society, October.

Margulis, N., 1989. The Intel 80860. *Byte*, 14(13):333–340, December.

Margulis, N., 1990. *i860 Microprocessor Architecture.* Osborne McGraw Hill, Berkeley, Cal., USA.

Markas, T., Royals, M., and Kanopoulos, N., 1990. On Distributed Fault Simulation. *Computer*, 23(1):40–52, January.

Masscomp., . *Performance Architecture Hardware.* Massachusetts Computer Corporation, Westford, MA, USA.

McDowell, C. E. and Helmbold, D. P., 1989. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December.

Merrow, T. and Henson, N., 1989. System Design for Parallel Computing. *High Performance Systems*, :36–44, January.

Metcalfe, R. M. and Boggs, D. R., 1976. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July.

Micrology., 1985. *VMEbus Specification Manual.* Micrology pbt, Inc, Tempe, Arizona, USA, revision c.1 edition, October.

Milenkovic, M., 1990. Microprocessor Memory Management Units. *IEEE Micro*, 10(2):70–85, April.

Motorola., 1984. *MC68020 32-Bit Microprocessor User's Manual.* Prentice-Hall.

Motorola., 1985. *MC68881 Floating-Point Coprocessor User's Manual.* Motorola.

Motorola., 1986. *MVME133 VMEmodule 32-Bit Monoboard Microcomputer User's manual.* Motorola Semiconductor Products Inc., Phoenix, Arizona, USA, first edition, September.

Mueller-Thuns, R. B., Saab, D. G., Damiano. R. F., and Abraham, J. A., 1989. Portable Parallel Logic and Fault Simulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 506–509, IEEE Computer Society, November.

Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H., 1990. Amoeba: A Distributed Operating System for the 1990s. *Computer*, 23(5):44–53, May.

Murai, S., Matsushita, H., and Enomoto, K., 1986. Logic Simulation Programs. In Hörbst, E., editor, *Logic Design and Simulation*, chapter 5, Elsevier Science, Amsterdam, The Netherlands.

Murakami, K., Irie, N., Kuga, M., and Tomita, S., 1989. SIMP (Single Instruction stream/Multiple instruction Pipelining): A Novel High-Speed Single-Processor Architecture. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*, pages 78–85, ACM SIGARCH, Computer Architecture News 17(3), June.

Murphy, E. E. and Voelcker, J., 1990. Systems Software. *IEEE Spectrum*, 27(1):38–40, January.

Newell, M. E. and Fitzpatrick, D. T., 1982. Exploitation of Hierarchy in Analyses of Integrated Circuit Artwork. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-1(4):192–200, October.

Newton, A. R. and Sangiovanni-Vincentelli, A. L., 1984. Relaxation-Based Electrical Simulation. *IEEE Transactions on Computer-Aided Design*, CAD-3(4):308–331, October.

Newton, A. R., 1979. Techniques for the Simulation of Large-Scale Integrated Circuits. *IEEE Transactions on Circuits and Systems*, CAS-26(9):741–749, September.

Newton, A. R., 1981. Timing, Logic and Mixed-mode Simulation for Large MOS Integrated Circuits. In Antognetti, P., Pederson, D. O., and De Man, H., editors, *Computer Design Aids for VLSI Circuits*, pages 19–112, Sijthoff & Noordhoff, Alphen aan den Rijn, The Netherlands.

Nickolls, J. R., 1990. The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer. In *Proceedings of the COMPCON Spring Conference*, pages 25–28, IEEE Computer Society, February.

Odent, P., Claesen, L., and De Man, H., 1989. Feedback loops and large subcircuits in the multiprocessor implementation of a relaxation based circuit simulator. In *Proceedings of the 26th Design Automation Conference (DAC)*, pages 25–30, ACM/IEEE, June.

Ohtsuki, T., editor. *Layout Design and Verification.* Volume 4 of *Advances in CAD for VLSI*, Elsevier Science, Amsterdam, The Netherlands.

Olson, T. M., 1989. Disk Array Performance in a Random IO Environment. *Computer Architecture News*, 17(5):71–77, September.

Overhauser, D., Hajj, I., and Hsu, Y., 1989. Automatic Mixed-Mode Timing Simulation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 84–87, IEEE Computer Society, November.

Patil, S. and Banerjee, P., 1990. A Parallel Branch and Bound Algorithm for Test Generation. *IEEE Transactions on Computer-Aided Design*, 9(3):313–322, March.

Patterson, D. A., Gibson, G., and Katz, R. H., 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the International Conference on Management of Data*, pages 109–116, ACM SIGMOD, June.

Piepho, R. S. and Wu, W. S., 1989. A Comparison of RISC Architectures. *IEEE Micro*, 9(4):51–62, August.

Preas, B. T. and Karger, P. G., 1988. Placement, Assignment and Floorplanning. In Preas, B. T. and Lorenzetti, M. J., editors, *Physical Design Automation of VLSI Systems*, chapter 4, Benjamin/Cummings, Menlo Park, CA, USA.

Preas, B. T. and Lorenzetti, M. J., editors. *Physical Design Automation of VLSI Systems*. Benjamin/Cummings, Menlo Park, CA, USA.

Quarterman, J. S., Silberschatz, A., and Peterson, J. L., 1985. 4.2BSD and 4.3BSD as Examples of the UNIX System. *ACM Computing Surveys*, 17(4):379–418, December.

Ramachandran, U., Solomon, M., and Vernon, M. K., 1990. Hardware Support for Interprocess Communication. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):318–329, July.

Ramamoorthy, C. V. and Li, H. F., 1977. Pipeline Architecture. *ACM Computing Surveys*, 9(1):61–101, March.

Rammig, F. J., 1986. Mixed Level Modelling and Simulation of VLSI Systems. In Hörbst, E., editor, *Logic Design and Simulation*, chapter 4, Elsevier Science, Amsterdam, The Netherlands.

Ranka, S., Won, Y., and Sahni, S., 1988. Programming a Hypercube Multicomputer. *IEEE Software*, 5(5):69–77, September.

Rashid, R. F. and Robertson, G. G., 1981. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64–75, ACM SIGOPS, December.

Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. J., and Chew, J., 1988. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–907, August.

Rau, B. R., Yen, D. W. L., Yen, W., and Towle, R. A., 1989. The Cydra 5 Departmental Supercomputer. *Computer*, 22(1):12–35, January.

Reddy, A. L. N. and Banerjee, P., 1989. An Evaluation of Multiple-Disk I/O Systems. *IEEE Transactions on Computers*, 38(12):1680–1690, December.

Reddy, A. L. N. and Banerjee, P., 1990. Design, Analysis, and Simulation of I/O Architectures for Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):140–151, April.

Reddy, A. L. N., Banerjee, P., and Abraham, S. G., 1988. I/O Embedding in Hypercubes. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 331–338, IEEE Computer Society, August.

Ritchie, D. M. and Thompson, K., 1974. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July.

Rose, J., 1988. Locusroute: A Parallel Global Router for Standard Cells. In *Proceedings of the 25th Design Automation Conference (DAC)*, pages 189–195, ACM/IEEE, June.

Rose, J. S., Snelgrove, W. M., and Vranesic, Z. G., 1988. Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing. *IEEE Transactions on Computer-Aided Design*, 7(3):387–397, March.

Rosenstiel, W. and Bergsträsser, T., 1986. Artificial Intelligence for Logic Design. In Hörbst, E., editor, *Logic Design and Simulation*, chapter 8, Elsevier Science, Amsterdam, The Netherlands.

Ross, F. E., 1986. FDDI - a Tutorial. *IEEE Communications*, 24(5):10–17, May.

Roth, J. P., 1966. Diagnosis of Automata Failures: A Calculus and a Method. *IBM Journal of Research and Development*, 10(4):278–291, July.

Roth, J. P., 1980. *Computer Logic, Testing and Verification.* Computer Science Press, Potomac, Maryland, USA.

Rubin, S. M., 1987. *Computer Aids for VLSI Design.* Addison-Wesley, Reading, Mass., USA.

Russell, C. H. and Waterman, P. J., 1987. Variations on UNIX for Parallel-Processing Computers. *Communications of the ACM*, 30(12):1048–1055, December.

Russell, R. M., 1978. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January.

Rutenbar, R. A., 1989. Simulated Annealing Algorithms: An Overview. *IEEE Circuits and Devices Magazine*, :19–26, January.

Rutenbar, R. A., Mudge, T. N., and Atkins, D. E., 1984. A Class of Cellular Architectures to Support Physical Design Automation. *IEEE Transactions on Computer-Aided Design*. CAD-3(4):264–278, October.

Sadayappan, P. and Visvanathan, V., 1988. Circuit Simulation on Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 37(12):1634–1642, December.

Saleh, R. A., Gallivan, K. A., Chang, M., Hajj, I. N., Smart, D., and Trick, T. N., 1989. Parallel Circuit Simulation on Supercomputers. *Proceedings of the IEEE*, 77(12):1915–1931, December.

Salem, K. and Garcia-Molina, H., 1986. Disk Striping. In *Proceedings of the International Conference on Data Engineering*, pages 336–342, IEEE Computer Society.

Sangiovanni-Vincentelli, A. L., 1981. Circuit Simulation. In Antognetti, P., Pederson, D. O., and De Man, H., editors, *Computer Design Aids for VLSI Circuits*, pages 19–112, Sijthoff & Noordhoff, Alphen aan den Rijn, The Netherlands.

Sargent, J. S. and Banerjee, P., 1989. A Parallel Row-Based Algorithm for Standard Cell Placement with Integrated Error Control. In *Proceedings of the 26th Design Automation Conference (DAC)*, pages 590–593, ACM/IEEE, June.

Scheffer, L. K. and Soetarman, R., 1985. Hierarchical Analysis of IC Artwork with User Defined Abstraction Rules. In *Proceedings of the 22nd Design Automation Conference (DAC)*, pages 293–298, ACM/IEEE, June.

Scheifler, R. W. and Gettys, J., 1986. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April.

Schindler, M., 1987. Demands on simulators escalate as circuit complexity explodes. *Electronic Design*, :11–32, October.

Sechen, C. and Sangiovanni-Vincentelli, A., 1985. The TimberWolf Placement and Routing Package. *IEEE Journal of Solid State Circuits*, SC-20(2):510–522, April.

Sechen, C., 1988. Chip-Planning, Placement, and Global Routing of Macro/Custom Cell Integrated Circuits Using Simulated Annealing. In *Proceedings of the 25th Design Automation Conference*, pages 73–80, ACM/IEEE, June.

Seiler, L., 1982. A Hardware Assisted Design Rule Check Architecture. In *Proceedings of the 19th Design Automation Conference (DAC)*, pages 232–238, ACM/IEEE, June.

Shing, M. T. and Hu, T. C., 1986. Computational Complexity of Layout Problems. In Ohtsuki, T., editor, *Layout Design and Verification*, chapter 8, Elsevier Science, Amsterdam, The Netherlands.

Siegel, H. J., 1985. *Interconnection Networks for Large-Scale Parallel Processing.* Lexington Books, Lexington, Mass., USA.

SIGDA., 1988. SIGDA/DATC Roundtable: Point Accelerators vs. General-Purpose Machines for Electronic CAD. *SIGDA Newsletter*, 18(1):33–38, March.

Skillicorn, D. B., 1988. A Taxonomy for Computer Architectures. *Computer*, 21(11):46–57, November.

Smith, J. M. and Maguire, Jr., G. Q., 1989. Measured Response Times for Page-Sized Fetches on a Network. *Computer Architecture News*, 17(5):48–54, September.

Smith, A. J., 1981. Input/Output Optimization and Disk Architectures: A Survey. *Performance and Evaluation*, :104–117.

Smith, S. P., Underwood, B., and Newman, J., 1988. An Analysis of Parallel Logic Simulation on Several Architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 65–68. IEEE Computer Society, August.

Smith, M. D., Johnson, M., and Horowitz, M. A., 1989. Limits on Multiple Instruction Issue. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 290–302, ACM, Computer Architecture News 17(2), April.

Smith, II, R. J., 1986. Fundamentals of Parallel Logic Simulation. In *Proceedings of the 23rd Design Automation Conference (DAC)*, pages 2–12, ACM/IEEE, June.

Solchenbach, K. and Trottenberg, U., 1988. SUPRENUM: System essentials and grid applications. *Parallel Computing*, 7:265–281.

Soulé, L. and Blank, T., 1988. Parallel Logic Simulation on General Purpose Machines. In *Proceedings of the 25th Design Automation Conference (DAC)*, pages 166–171. ACM/IEEE, June.

Soulé, L. and Gupta, A., 1989. Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation. In *Proceedings of the 26th Design Automation Conference (DAC)*. pages 81–86, ACM/IEEE, June.

Stallings, W., 1988. Reduced Instruction Set Computer Architecture. *Proceedings of the IEEE*, 76(1):38–55. January.

Stankovic, J. A., 1988. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, October.

Stenström, P., 1988. Reducing Contention in Shared-Memory Multiprocessors. *Computer*, 21(11):26–37, November.

Stenström, P., 1990. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, June.

Stevens, S. N. and McCabe, S., 1985. An Integrated Approach for Hierarchical Verification of VLSI Mask Artwork. *IEEE Journal of Solid State Circuits*, SC-20(2):501–509, April.

Stone, H. S., 1987. *High-Performance Computer Architecture.* Addison Wesley, Reading, Mass., USA.

Stumm, M. and Zhou. S., 1990. Algorithms Implementing Distributed Shared Memory. *Computer*, 23(5):54–64, May.

Subramanian, K. and Zargham, M. R., 1990. Distributed and Parallel Demand Driven Logic Simulation. In *Proceedings of the 27th Design Automation Conference (DAC)*, pages 485–490, ACM/IEEE, June.

Sugla, B., Edmark, J., and Robinson, B. S., 1989. An Introduction to the CAPER Concurrent Application Programming Environment. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, IEEE Computer Society, August.

Swan, R. J., Fuller, S. H., and Siewiorek, D. P., 1977. Cm - A modular, multi-microprocessor. In *AFIPS Conference Proceedings*, pages 637–644.

Szymanski, T. G. and Van Wyk, C. J., 1985. Goalie: A Space Efficient System for VLSI Artwork Analysis. *IEEE Design & Test*, 2(3):64–72, June.

Szymanski, T. G. and Van Wyk, C. J., 1988. Layout Analysis and Verification. In Preas, B. T. and Lorenzetti, M. J., editors, *Physical Design Automation of VLSI Systems*, chapter 8, Benjamin/Cummings, Menlo Park, CA, USA.

Takasaki, S., Sasaki, T., Nomizu, N., Koike, N., and Ohmori, K., 1987. Block-Level Hardware Logic Simulation Machine. *IEEE Transactions on Computer-Aided Design*, CAD-6(1):46–54, January.

Tam, M., Smith, J. M., and Farber, D. J., 1990. *A Survey of Distributed Shared Memory Systems.* Technical Report, Distributed Systems Laboratory, University of Pennsylvania, Philadelphia, PA, USA, March.

Tanenbaum, A. S. and van Renesse, R., 1985. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):419–470, December.

Tanenbaum, A. S., van Staveren, H., Keizer, E. G., and Stevenson, J. W., 1983. A Practical Tool Kit for making Portable Compilers. *Communications of the ACM*, 26(9):654–660, September.

Terman, C. J., 1987. Simulation Tools for VLSI. In Fichtner, W. and Morf, M., editors, *VLSI CAD Tools and Applications*, chapter 3, Kluwer, Norwell, Mass., USA.

Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H., 1988. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August.

Thadhani, A. J., 1981. Interactive user productivity. *IBM Systems Journal*, 20(4):407–423.

Thakkar, S., Gifford, P., and Fielland, G., 1988. The Balance Multiprocessor System. *Computer*, 21(2):57–69, February.

Tonkin, B. A., 1990. Circuit Extraction on a Message Based Multiprocessor. In *Proceedings of the 27th Design Automation Conference*, pages 260–265, ACM/IEEE, June.

Tredennick, N., 1987. Trends in Commercial VLSI Microprocessor Design. In Fichtner, W. and Morf, M., editors, *VLSI CAD Tools and Applications*, chapter 10, Kluwer, Norwell, Mass., USA.

Tucker, L. W. and Robertson, G. G., 1988. Architecture and Applications of the Connection Machine. *Computer*, 21(8):26–38, August.

van Brunt, N., 1983. The Zycad Logic Evaluator and its Application to Modern System Design. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 232–233, IEEE Computer Society, October.

van Renesse, R., Tanenbaum, A. S., and Wilschut, A., 1989. The Design of a High-Performance File Server. In *Proceedings of the 9th IEEE International Conference on Distributed Computing Systems*, pages 22–27, IEEE Computer Society. June.

van Renesse, R., van Staveren, H., and Tanenbaum, A. S., 1989. Performance of the Amoeba Distributed Operating System. *Software - Practice and Experience*. 19:223–234, March.

Vaughan, F. A., Marlin, C. D., and Barter, C. J., 1988. A Distributed Operating System Kernel for a Closely-Coupled Multiprocessor. *The Australian Computer Journal*, 20(2):58–64, May.

Vitányi, P. M. B., 1988. Locality, Communication, and Interconnect Length in Multicomputers. *SIAM Journal on Computing*, 17(4):659–672, August.

Waltz, D. L., 1987. Applications of the Connection Machine. *Computer*, 20(1):85–97, January.

Warren, C., 1990. Micro Standards: The Scalable Coherent Interface. *IEEE Micro*, 10(3):80–82, June.

Wasserman, H. J., Simmons, M. L., and Lubeck, O. M., 1988. The performance of minisupercomputers: Allian FX/8, Convex C-1, and SCS-40. *Parallel Computing*, 8:285–293.

Waterman, P. J., 1987. On Parallel Circuit Simulation. *VLSI Systems Design*. :56–61. July.

Webber, D. M. and Sangiovanni-Vincentelli, A., 1987. Circuit Simulation on the Connection Machine. In *Proceedings of the 24th Design Automation Conference*, pages 108–113, ACM/IEEE, June.

Weiss, S., 1989. Scalar Supercomputer Architecture. *Proceedings of the IEEE*, 77(12):1970–1982. December.

Widdowson, R. and Ferguson, K., 1988. Parallel Polygon Operations Using Loosely Coupled Workstations. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 276–279, IEEE Computer Society, November.

Wiley, P., 1987. A parallel architecture comes of age at last. *IEEE Spectrum*, :46–50, June.

Williams, T. W., 1981. Design for Testability. In Antognetti, P., Pederson, D. O., and De Man, H., editors, *Computer Design Aids for VLSI Circuits*, pages 359–416. Sijthoff & Noordhoff, Alphen aan den Rijn, The Netherlands.

Wolf, W. H. and Dunlop, A. E., 1988. Symbolic Layout and Compaction. In Preas, B. T. and Lorenzetti, M. J., editors, *Physical Design Automation of VLSI Systems*, chapter 6, Benjamin/Cummings, Menlo Park, CA, USA.

Wong. C. P. and Fiebrich, R. D., 1987. Simulated Annealing-Based Circuit Placement Algorithm on the Connection Machine System. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 78–82, IEEE Computer Society, October.

Wu, K. and Fuchs, W. K., 1990. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, April.

Wu, M. and Gajski, D. D., 1989. Computer-Aided Programming for Message Passing Systems: Problems and a Solution. *Proceedings of the IEEE*, 77(12):1983–1991, December.

Wu, M. and Gajski, D. D., 1990. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July.

Yang, G., 1990. PARASPICE: A Parallel Circuit Simulator for Shared-Memory Multiprocessors. In *Proceedings of the 27th Design Automation Conference (DAC)*, pages 400–405. ACM/IEEE, June.

Yeh, D. C. and Rao, V. B., 1988. Partitioning Issues in Circuit Simulation on Multiprocessors. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. pages 300–303, IEEE Computer Society, November.

Young. M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., 1987. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 63–76, ACM SIGOPS, Operating Systems Review, November.

Zargham, M. R., 1988. Parallel Channel Routing. In *Proceedings of the 25th Design Automation Conference (DAC)*, pages 128–133, ACM/IEEE, June.