## CHIENG-FAI LIM

B.S., University of Illinois, 1990


Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991


DISPLAY 92220061/2
92N20061*H ISSUE 11 PAGE 1855 CATEGORY 61 RPTH: NASA-CR-189925 NAS 1.26:189925 CXTH: NAG1-613 91/00/00 80 PAGES UNCLASSIFIED DOMETT
UTTL: A parallel algorithm for multi-level logic synthesis using the transduction method TLSP: M.S. Thesis
ACTH: A/LTM, CHIEVG-FAI
CDRP: Illinois Univ., Urtana-Charpaign. CSS: (Coordinated Science Lab.)
SAP: Avail: CASI HC A05/ NF A01
COO: UNITE STATES
WIS: /*ALGORITHM/ + COMPUTER SYSTEMS PERFORMACE/*MEORY (COMPUTERS)/* MLTIPROCESSISG (COYPUTERS)/*OPTIITZATIOV/ $\times$ PARALLEL PROCESSING (COMPUTERS)
 PaRTITIONS (MATHEAIICS)/ SLDSTITUTES/ TRAVSFERRIVG
ABA: Author
ABS: The Transduction Method has been shown to be a powerful tool in the optimization of multilevel netwoms. Many tools such as the SYLON synthesis system (XCO), (CH189), (LiDO) have been developed based on this method. A parallel implementation is presented of SYLOX-XTRAYS (XY89) on an eight processor Encore : Wlltirax shared memory multiprocessor. It minimizes multilevel networks consisting of simple gates through parallel pruning, gate substitution, gate merging, generalized gate substitution, and gate input reduction. This implementation, called Parallel

# A PARALLEL ALGORITHM FOF MULTI-LEVEL <br> LOGIC SYNTHESIS USING THE TRANSDUCTION METHOD 

BY<br>CHIENG-FAI LIM<br>B.S., University of Illinois, 1990

THESIS
Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois


#### Abstract

The Transduction Method has been shown to be a powerful tool in the optimization of multi-level networks. Many tools such as the SYLON synthesis system [X90], [CM89], [LM90] have been developed based on this method. In this paper, we present a parallel implementation of SYLON-XTRANS [XM89] on an eight-processor Encore Multimax shared-memory multiprocessor. It minimizes muli-level networks consistings of simple gates through parallel pruning, gate substitution, gate merging, generalized gate substitution, and gate input reduction. This implementation, called Parallel TRANSduction (PTRANS), also uses partitioning to break large circuits up and performs inter- and intra-partition dynamic load balancing. With this, we are able to achieve good speedups and high processor efficiencies without sacrificing the resulting circuit quality.


## ACKNOWLEDGEMENTS

I am most grateful for the constant advice and support of my advisor, Professor Prithviraj Banerjee, who has made the completion of this thesis possible.

I would like to thank Professor Saburo Muroga and his students who have shared their valuable experiences with me. I would also like to thank the students and staff members of the Center for Reliable and High-Performance Computing who have been a great source of help. Specifically, I am thankful for Kaushik De for his ideas and assistance.

Finally, I would like to thank my fellow graduate students for making my stay in this country a precious experience.

## TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION ..... 1
1.1. Motivation for Parallel CAD Algorithms ..... 1
1.2. Two-level and Multi-level Logic Synthesis ..... 2
1.3. Related Work on Parallel Logic Synthesis ..... 2
1.3.1. Parallel ESPRESSO ..... 3
1.3.2. Parallel Kernel Extraction ..... 4
1.3.3. Parallel Tautology Chẹcking ..... 5
1.4. Thesis Outline ..... 5
CHAPTER 2. REVIEW OF THE TRANSDUCTION METHOD ..... 7
2.1. Terminology and Notations ..... 7
2.2. Maximum Set of Permissible Functions ..... 11
2.3. Compatible Set of Permissible Functions ..... 15
2.4. Pruning ..... 17
2.5. Gate Substitution ..... 20
2.6. Gate Merging ..... 21
2.7. Generalized Gate Substitution ..... 24
2.8. Gate Input Reduction ..... 26
CHAPTER 3. PARALLEL IMPLEMENTATION OF SYLON-XTRANS ..... 29
3.1. General Overview ..... 29
3.2. Binary Decision Diagrams ..... 30
3.3. Partitioning Algorithm ..... 37
3.4. Program Model ..... 38
3.5. Discussion of Number of Partitions ..... 41
3.6. Parallel Evaluation of Functions and CSPFs c: Gates ..... 42
3.7. Parallel Pruning ..... 46
3.8. Parallel Gate Substitution ..... 48
3.9. Parallel Gate Merging ..... 51
3.10. Parallel Generalized Gate Substitution/Gate Input Reduction ..... 54
3.11. Ordering of Search-Spaces ..... 57
CHAPTER 4. EXPERIMENTAL RESULTS ..... 59
4.1. Overview of Experiments ..... 59
4.2. Circuit Degradation with Number of Processo:s ..... 60
4.3. Efficiency of Intra-Partition Load Balancing ..... 61
4.4. Efficiency of Inter-Partition Load Balancing ..... 63
4.5. Comparison among MIS 2.1, SYLON-XTRANS, and PTRANS ..... 66
CHAPTER 5. CONCLUSIONS ..... 70
REFERENCES ..... 72

## CHAPTER 1.

## INTRODUCTION

### 1.1. Motivation for Parallel CAD Algorithms

Computer Aided Design (CAD) algorithms alvays face the conflict between the need to produce superior quality results and the need to shorten the long processing time they require. Many problems in VLSI CAL are NP-complete [GJ79], hence determining the optimum solutions to these problems can take extraordinary amounts of CPU time. Hence, heuristics are used to reduce their complexities so that the results can be delivered within a reasonable amount of time.

To reduce the runtimes of CAD tools, a simple way is to execute them on faster uniprocessor machines. However, this is no longer feasible as we are approaching an upper bound on the speed of the processors that can be made with current technology. This problem has led to more attention being focused on parallel machines.

With today's increasing availability and perfornnance of parallel machines, a new direction has been created for parallel processing of $C A D$ algorithms. Many of the $C A D$ applications have a high degree of inherent parallelism. There is a bright future in the integration of new parallel programming paradigms, parallel architectures, and CAD algorithms so as to provide users with a shorter turnaround time.

### 1.2. Two-level and Multi-level Logic Synthesis

Automation of logic synthesis tools is becoming increasingly important as the number of logic gates in VLSI chips gets larger. In the past, many studies were devoted to realizing combinational logic functions with 2-level networks using PLA's. Many efficient algorithms such as ESPRESSO [B84] and PMIN [C87] have been developed.

Unfortunately, many combinational logic functions can be more efficiently realized with multi-level networks in terms of compactness, cost, and speed. Many tools have also been developed for multi-level logic synthesis. SOCRATES [GBGH86] and MIS [BRSW87] are among them. In the early 70's, the Transduction Method was developed at University of Illinois. This involves the concept of permissible functions, which is also regarded frequently as observability don't-cares. Based on this method, SYLONXTRANS [X90], SYLON-DREAM [CM89], and SYLON-REDUCE [LM90] have been developed. They have shown that the Transduction Method is a powerful tool in the optimization of multi-level circuits.

### 1.3. Related Work on Parallel Logic Synthesis

With increasing accessibility of parallel machines, there have been many studies on parallel CAD algorithms. This section review; some of such work including Galivanche's parallel ESPRESSO [G86], Zipfel's parallel kernel extractor [Z91] and Hatchel's parallel tautology checking [HMJ88].

### 1.3.1. Parallel ESPRESSO

In ESPRESSO, there are three main procedures called Complement, Expand, and Reduce. The section describes their parallelization processes described by Galivanche [G86].

To compute the complement of a given function, the Complement procedure recursively decomposes it into two sub-functions along a splitting variable until a single term is reached. In the parallel version, a new process is created at each level of the recursion so that the two sub-functions can be handled simultaneously. This creation of processes stops when the number of processes created equals the number of processors available.

The Expand procedure generates a limited set of prime cubes of a given function. The set of cubes under consideration are maintained in a list. Each cube is expanded with the objective of covering other cubes in the lis'. In the parallel algorithm, cubes are expanded in parallel. However, duplicated cubes can be created. To minimize this redundant work, periodic checks are made to halt duplicated work. The procedure also terminates with a final clean-up phase to remove the auplicated cubes.

The third procedure, Reduce, tries to obtain a minimal number of cubes covering a given function so that any further reduction would change the function. Although most of the cubes can be reduced simultaneously, a process could be reducing a cube $C_{i}$ thinking that it is covered by another cube $C_{j}$ without knowing that $C_{j}$ is also being currently reduced by another process. The solution tc this problem is to assume that all
other cubes currently being reduced do not exist. Although this gives correct outputs, it affects the quality of the final results.

Galivanche achieved linear speedup in completion time with slight degradations in the resulting qualities of the generated PLA's with these algorithms.

### 1.3.2. Parallel Kernel Extraction

Zipfel [Z91] has implemented a parallel version of the kernel extraction procedure used in MIS during algebraic factorization [BRSW87].

First, the kernel-cube matrix is built in parallel. This is also executed in parallel with the formation of the Boolean representation of a node since they are independent. After building the kernel-cube matrix, the next phase performs the actual extraction. In parallel, each process creates its own local partition of the kernel-cube matrix and uses it to perform any extraction from the global network. These partitions are generated in parallel as well.

With its own partition of the kernel-cube matriz, a process then proceeds to look for kernel intersections that are extractable. If the value of a kernel intersection is greater than zero, a new node is then created and exclusively substituted into the Boolean network. When a process has exhausted its partition, it waits until all of the other processes have exhausted theirs before repeating the Kernel-cube matrix-building algorithm again. With this parallel algorithm, Zipfel was able to obtain slight improvements in the minimality of the circuits tested. However, the low speedups he
achieved showed that MIS is very difficult to parallelize.

### 1.3.3. Parallel Tautology Checking

Hatchel's parallel tautology checking algorithm [HJM88] uses the parallelism of a serial divide-and-conquer algorithm. The serial algorithm recursively divides the function into smaller partitions until the function to be checked is sufficiently small. In the parallel version, a process is created for each sub-function if a partition is found to be complicated enough. Each process waits for all of its children (if any) to report back before terminating. With this tree-structured computation, good speedup has been achieved.

### 1.4. Thesis Outline

This thesis describes a parallel implementation of the Transduction Method of multi-level logic synthesis on a shared-memory machine, the Encore Multimax computer. The implementation, called PTRANS (Parallel TRANSduction), is based on SYLON-XTRANS [X90], [XM89].

Scalability has been a problem in the parallelization process. In order to maintain high processor utilization when the number of processors increases, the circuit to be minimized has to be large. Unfortunately, the amount of physical memory available places an upper bound on the size of the circuit to be minimized.

To solve this problem, large circuits have to be partitioned. The partitioning algorithm tries to retain the don't-cares within a partition. Clearly, the minimization of the partitions can be performed in parallel. However: although two partitions may be of the same size in terms of the number of gates and connections, the time required to minimize each of them could be different due to differences in their functional complexities.

This thesis describes how the partitions can be minimized simultaneously with both inter- and intra-partition parallelism being handled by dynamic load balancing. The organization of this report is as follows. In Chapter 2, the basic concepts of the Transduction Method is given. It also provides sone background information on the permissible functions and the transformation and reduction procedures found in SYLON-XTRANS. The parallelization of these procedures and the implementation of dynamic load balancing is presented in Chapter 3. In Chapter 4, some experimental results achieved with PTRANS are reported, followed by a conclusion in Chapter 5.

## CHAPTER 2.

## REVIEW OF THE TRANSDUCTION METHOD

SYLON-XTRANS is an extension to the original Transduction Method in [MK89] so that it can minimize multi-level circuits consisting of AND, OR, NAND and NOT gates in addition to NOR gates. It contains four main procedures, namely, pruning, gate substitution, gate merging, and combined generalized gate substitution/gate input reduction. Each of these procedures is basically an iterative improvement algorithm that keeps transforming and reducing a circuit until no further improvement can be made. The transformations can be applied to a circuit in any order. However, formal proofs of the transformations are omitted in this thesis for simplicity. They can be found in [XM89] and [X90]. For the ease of translating into binary decision diagrams (BDDs) which are actually implemented in PTRANS, the transformations are explained using the vector notation.

### 2.1. Terminology and Notations

In this thesis, we will consider only cycle-free multi-level circuits consisting of AND, OR, NAND, NOR and NOT gates. Let $n$ be the number of primary inputs, $m$ be the number of primary outputs, and $g$ be the number of gates in a multi-level circuit. Let $X=\left\{x_{1}, x_{2}, \ldots, x_{n}\right\}$ be the set of input variables and $Z=\left\{z_{1}, z_{2}, \ldots, z_{m}\right\}$ be the set of output variables of the circuit. In addition, let $V=\left\{v_{1}, v_{2}, \ldots, v_{g}\right\}$ be the set of
gates in the circuit, and $C=\left\{c_{i j}\right\}$ be the set of connections where $c_{i j}$ connects the output of gate $\mathbf{v}_{\mathbf{i}}$ to an input of gate $\mathbf{v}_{\mathbf{j}}$.

A circuit can be viewed as a graph consisting of gates arranged in levels. The level of a gate in a circuit can be defined either from the primary inputs or the primary outputs. Formally, the level of a gate with respect to the primary inputs is defined as :

1) 0 if the gate is a primary input, or
2) $1+$ the maximum level among its immediate predecessors.

The level of a gate with respect to the primary outputs is similarly defined as :

1) 0 if the gate is a primary output, or
2) $1+$ the maximum level among its immediate successors.

The levelizing procedure for a circuit can be foand in [PBP89]. An example of a circuit whereby the gates are arranged according to their levels is shown in Figure 2.1.


Figure 2.1. An example of a levelized circuit.

A gate $v_{i}$ is an immediate predecessor of $\mathbf{v}_{\mathrm{j}}$ if there exists a connection $\mathrm{c}_{\mathrm{ij}}$. Conversely, $v_{j}$ is an immediate successor of $v_{i}$ if $c_{i j}$ exists. Let $\operatorname{IP}\left(v_{i}\right)$ and $\operatorname{IS}\left(v_{i}\right)$ be the set of all immediate predecessors and immediate successors of the gate $v_{i}$ respectively. When there is a sequence of gates $v_{k 1}, v_{k 2}, \ldots . v_{k t}$ such that $v_{k b+1} \in I S\left(v_{k b}\right)$ for all $b$ $=1,2, \ldots t-1$, then $v_{k t}$ is a successor of $v_{k 1}$. Similariy defined, $v_{k 1}$ is a predecessor of $\mathrm{v}_{\mathrm{kt}}$. Let $\mathrm{P}\left(\mathrm{v}_{\mathrm{i}}\right)$ and $\mathrm{S}\left(\mathrm{v}_{\mathrm{i}}\right)$ denote the set of predecessors and successors of the gate $\mathrm{v}_{\mathrm{i}}$ respectively. The gate $v_{i}$ is said to have a reconvergent fanout (or is reconvergent) if there exist two distinct gates $\mathrm{v}_{\mathrm{k} 1}, \mathrm{v}_{\mathrm{k} 2} \in \mathrm{IS}\left(\mathrm{v}_{\mathrm{i}}\right)$ such that $\mathrm{S}\left(\mathrm{v}_{\mathrm{k} 1}\right) \cap \mathrm{S}\left(\mathrm{v}_{\mathrm{k} 2}\right) \neq \varnothing$.

A function realized at a gate is the set of values output by the gate in a circuit for all combinations of the input variables. This is also very frequently referred to as the function at the gate for short. The function at a gate $v_{i}, f\left(v_{i}\right)$, can be expressed as a vector of Boolean values. For example, if $n=3$ and $v_{i}$ is an AND gate with $x_{1}, x_{2}$ and $x_{3}$ as its input, where $\mathrm{x}_{1}=(01010101), \mathrm{x}_{2}=(00110011)$, and $\mathrm{x}_{3}=(00001111)$, then $f\left(v_{i}\right)=(00000001)$. Also, if $G$ is a Boolean vector, iet $G^{(d)}$ be the dth value in G. It does not matter if the first value is the leftmost or rightmost bit of a vector as long as this remains consistent. The value of $d$ can range from 1 to $2^{n}$ inclusive. This is a more convenient way of representing the truth table. In the Transduction Method, connections are often treated as gates. Hence, the function at a gaie is also extended to cover that of a connection, which is defined by $f\left(c_{i j}\right)=f\left(v_{i}\right)$.

The function at a gate $v_{i}$ can sometimes be expressed not only in terms of the input variables but also as the function at some other gate $\mathrm{v}_{\mathrm{k}}$ in the circuit. This is denoted by $f\left(v_{i} \mid v_{k}\right)$ and is called the function at $v_{i}$ with respect to $v_{k}$. In this case, the
gate $\mathrm{v}_{\mathrm{k}}$ is treated just as it is an input variable ignoring the functions at its input connections.

Very frequently, the function at a gate can be changed without affecting the function at the primary outputs. A permissible function at a gate is a function which the output of a gate can be for this purpose. For exar.ple, in Figure 2.2, for $n=3, x_{1}=$ (01010101), $\mathrm{x}_{2}=(00110011), \mathrm{x}_{3}=(00001111), \mathrm{f}\left(\mathrm{v}_{1}\right)=(11101110)$ and $\mathrm{z}_{1}=\mathrm{f}\left(\mathrm{v}_{2}\right)=$ (10111011). However, if $f\left(v_{1}\right)$ is changed to ( 01101110 ), $z_{1}$ is still unchanged. Hence, ( 01101110 ) is a permissible function of $\mathrm{v}_{1}$.

The vector ( 01100110 ) is another permissible function of $v_{1}$. To represent these two permissible functions collectively, a don't-care value '*' is used. This is used to mean either a ' 0 ' or ' 1 ' value. Hence, ( $0110 * 110$ ) represents both ( 01101110 ) and (01100110). A collection of permissible functions is known as a set of permissible functions (SPF), of which two special forms are the maximum set of permissible functions and compatible set of permissible functions. These are explained in greater details in Sections 2.2 and 2.3.


Figure 2.2. An example of a permissible function.

### 2.2. Maximum Set of Permissible Functions

As the name suggests, the maximum set of permissible functions (MSPF) of a gate in a circuit is the set that contains all possible permissible functions of the gate. [MK89] shows how the MSPFs for gates and connections in a multi-level circuit containing only NOR gates are calculated. [X90] extends this to OR, AND, NAND and NOT gates.

In [X90], the methods of calculating MSPF's are described using the on-set/off-set notation since it uses the sum-of-products (SOP) form to represent Boolean and permissible functions. However, PTRANS uses tinary decision diagrams (BDDs) [B86]. As it is convenient to translate bit vectors into BDDs, the methods of calculating MSPFs and CSPFs are shown in this thesis using the vector notation instead. This is similar to that used in [MF89]. Section 4.2 shows how the vector notation can be translated into BDD representation.

Before formally describing the methods of computing MSPFs, an example is given here. From Figure 2.2, we have $x_{1}=(01010101), f\left(v_{1}\right)=(11101110)$, and $f\left(v_{2}\right)=$ (10111011). Since the output $z_{1}=f\left(v_{2}\right)$ must remain constant, $\operatorname{MSPF}\left(v_{2}\right)=f\left(v_{2}\right)=$ (10111011). Let the first bits of the vectors to be the leftmost bits. Considering these bits, $x_{1}=0$ and $f\left(v_{2}\right)=1$. Since $v_{2}$ is a NAND-gate, and $x_{1}$ is $0, f\left(v_{2}\right)$ is always 1 regardless of the value of $f\left(v_{1}\right)$. Hence, the first bit of the MSPF of $v_{1}$ is *. Similarly, the rest of MSPF bits of $v_{1}$ can be computed, and this vector is found to be $(* 1 * 0 * 1 * 0)$.

The ways of computing the MSPFs of gates and of connections are different. To show how the MSPF of a connection $c_{i j}$ is computed, consider a portion of a circuit which contains $\mathrm{c}_{\mathrm{ij}}$ as shown in Figure 2.3.

Suppose the functions at all of the connections $c_{x j}$ for $1 \leq x \leq k$ and at $v_{j}$ are known. Let the $\operatorname{MSPF}$ of $\mathbf{v}_{\mathbf{j}}$ be $\operatorname{MSPF}\left(\mathbf{v}_{\mathbf{j}}\right)$ and suppose that it is known too. If $\mathbf{v}_{\mathbf{j}}$ is a NOR gate, the dth bit of the vector $\operatorname{MSPF}\left(c_{i j}\right), \operatorname{MSPF}^{(d)}\left(c_{i j}\right)$, is then given by :

$$
\begin{equation*}
\operatorname{MSPF}^{(d)}\left(c_{i j}\right)=F^{(d)} \#_{N O R} \operatorname{MSPF}^{(d)}\left(v_{j}\right) \tag{E2.1}
\end{equation*}
$$

where the operator $\#_{N O R}$ is defined in Table 2.1 and $F=U_{1 \leq x \leq k, x \neq i} f\left(c_{x j}\right), U$ being the normal Boolean OR operator. Similarly, for the cases in which $v_{j}$ is an OR, AND, or NAND gate, $\operatorname{MSPF}^{(d)}\left(\mathrm{c}_{\mathrm{ij}}\right)$ is given by equations E2.2, E2.3 and E2.4 respectively. The vector $G$ in $E 2.3$ and $E 2.4$ is $\cap_{1 \leq x \leq k, x \neq i} f\left(c_{x_{i}}\right)$, where $\cap$ is the Boolean AND
 2.4 respectively. The '-' sign in these tables means that those situations will never be encountered.

$$
\begin{align*}
& \operatorname{MSPF}^{(d)}\left(c_{i j}\right)=F^{(d)} \#_{O R} \operatorname{MSPF}^{(d)}\left(v_{j}\right)  \tag{E2.2}\\
& \operatorname{MSPF}^{(d)}\left(c_{i j}\right)=G^{(d)} \#_{A N D} \operatorname{MSPF}^{(d)}\left(v_{j}\right)  \tag{E2.3}\\
& \operatorname{MSPF}^{(d)}\left(c_{i j}\right)=G^{(d)} \#_{N A N D} \operatorname{MSPF}^{(d)}\left(v_{j}\right) \tag{E2.4}
\end{align*}
$$



Figure 2.3. Calculating the MSPF of a connection.

A special case arises when $v_{j}$ is a NOT gate. $\operatorname{MSPF}^{(d)}\left(c_{i j}\right)$ is then computed simply by E 2.5 where $\sim$ is the COMPLEMENT operitor.

$$
\begin{equation*}
\operatorname{MSPF}^{(d)}\left(c_{i j}\right)=\sim \operatorname{MSPF}^{(d)}\left(v_{j}\right) \tag{E2.5}
\end{equation*}
$$

The evaluation of the MSPF of a gate is slightly more complicated. Consider a gate $\mathrm{v}_{\mathrm{i}}$ as shown in Figure 2.4 and having $\mathrm{c}_{\mathrm{i} 1}, \mathrm{c}_{\mathrm{i} 2}, \ldots \mathrm{c}_{\mathrm{ik}}$ connected to its output terminal.

If the gate $v_{i}$ is not a reconvergent gate, its MSPF is given by E2.6 where $* \cap 1=$ $1 \cap^{*}=1$ and $* \cap 0=0 \cap^{*}=0$ in addition to the normal properties of $\cap$ on the domain \{0,1\}.

| Table 2.1 |  |  |  |  | Table 2.2 |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| \# NOR |  | MSPF ${ }^{\text {(d) }}$ |  |  | \# OR |  | MSPF ${ }^{\text {(d) }}$ |  |  |
|  |  | 0 | 1 | * |  |  | 0 | 1 | * |
| $\mathrm{F}^{(d)}$ | 0 | 1 | 0 | * | $\mathrm{F}^{(\mathrm{d})}$ | 0 | 0 | 1 | * |
|  | 1 | * | - | * |  | 1. | - | * | * |
| Table 2.3 |  |  |  |  | Table 2.4 |  |  |  |  |
| \#NAND |  | MSPF $^{\left({ }^{\text {(d) }}\right.}$ |  |  | \#AND |  | $\mathrm{MSPF}^{\text {(d) }}$ |  |  |
|  |  | 0 | 1 | * |  |  | 0 | 1 | * |
| $\mathrm{G}^{(d)}$ | 0 | $\because$ | * | * | $\mathrm{G}^{(\mathrm{d})}$ | 0 | * | - | * |
|  | 1 | 1 | 0 | * |  | 1 | 0 | 1 | * |

Tables 2.1 through 2.4.



Figure 2.4. Calculating the MSPF of a gate.

$$
\begin{equation*}
\operatorname{MSPF}^{(d)}\left(v_{i}\right)=\cap_{1 \leq x \leq k} \operatorname{MSPF}^{(d)}\left(c_{i x}\right) \tag{E2.6}
\end{equation*}
$$

However, if $\mathbf{v}_{\mathbf{i}}$ is reconvergent, it is then treated as an input variable and all the primary outputs are evaluated with respect to $\dot{v}_{i}$. Using Shannon's Expansion, the function at every output $\mathrm{z}_{\mathrm{j}}$ can then be expressed as:

$$
f\left(z_{j} \mid v_{i}\right)=f\left(v_{i}\right) \cap P_{j} \cup \sim f\left(v_{i}\right) \cap Q_{j}
$$

where $P_{j}$ and $Q_{j}$ are some functions expressed in terms of the primary inputs only. $\operatorname{MSPF}{ }^{(d)}{ }_{j}\left(v_{i}\right)$, which is the MSPF of $v_{i}$ due to $z_{j}$, can then be computed using the following algorithm :

If $f\left(v_{i}\right)^{(d)}=1$ and $Q^{(d)}{ }_{j}=0$ then $\operatorname{MSPF}^{(d)}{ }_{j}\left(v_{i}\right)=1$
else if $f\left(v_{i}\right)^{(d)}=0$ and $Q^{(d)}{ }_{j}=1$ then $\operatorname{MSPF}^{(d)}{ }_{j}\left(v_{i}\right)=1$
else if $f\left(v_{i}\right)^{(d)}=1$ and $P^{(d)} j=0$ then $\operatorname{MSPF}^{(d)} j_{j}\left(v_{i}\right)=0$
else if $f\left(v_{i}\right)^{(d)}=0$ and $P^{(d)}{ }_{j}=1$ then $\left.\operatorname{MSPF}^{(d)} \underset{j}{ } \dot{v_{i}}\right)=0$
else $\operatorname{MSPF}^{(\mathrm{d})}{ }_{j}\left(\mathrm{v}_{\mathbf{i}}\right)=*$.
The final value for $\operatorname{MSPF}^{(d)}\left(v_{i}\right)$ is then the intersection of $\operatorname{MSPF}^{(d)}{ }_{j}\left(v_{i}\right)$ for $1 \leq j$ $\leq \mathrm{m}$.

To explain the correctness of this algorithm, suppose $f^{(d)}\left(v_{i}\right)=1$ and $Q^{(d)}{ }_{j}=0$. Hence, $f^{(d)}\left(z_{j} \mid v_{i}\right)=P^{(d)}{ }_{j}$, which could be either 1 or 0 . Therefore, MSPF ${ }^{(d)}{ }_{j}\left(v_{i}\right)$ must be 1 so as to allow $P^{(d)}$ to propagate to $f^{(d)}\left(z_{j} \mid v_{i}\right)$. This argument is similar for the other three cases. ${ }^{\circ}$

Knowing how the MSPF of a connection and gate can be calculated, the MSPFs of all the gates and connections can then be calculated by first setting the MSPF of each
primary output to be the same as the function at the output gate and then compute the rest of the MSPFs from the outputs towards the primary inputs.

### 2.3. Compatible Set of Permissible Functions

As the MSPF of a gate contains the largest set of permissible functions associated with it, this set also contains the largest observability don't-care set [SB90] for the gate. The observability don't-care set of a gate is the set of input values with which the gate's output is not observable through the primary oxtputs. However, as seen from the previous section, the computation of MSPF could be time-consuming, especially when a circuit has many reconvergent gates. In addition, the MSPFs of all the gates and connections in a circuit have to be recomputed each time the circuit is transformed and reduced. Therefore, to reduce the amount of processing time required, compatible sets of permissible functions (CSPF) for gates and connections are more frequently used.

A compatible set of permissible functions. is a subset of the MSPF and is computed based on some ordering of the connections in a circuit. Although the don'tcare set associated with a CSPF is often smaller than that with the MSPF, the quality of the resulting circuit minimized based on CSPF usually does not suffer too badly and the processing time required is dramatically reduced.

CSPFs are computed similar to MSPFs. Referring to Figure 2.3 again, the CSPF for the connection $\mathrm{c}_{\mathrm{ij}}$ is given by :

$$
\begin{align*}
& \operatorname{CSPF}^{(d)}\left(c_{i j}\right)=F^{(d)} \#_{N O R}^{\operatorname{CSPF}^{(d)}\left(v_{j}\right)}  \tag{E2.7}\\
& \operatorname{CSPF}^{(d)}\left(c_{i j}\right)=F^{(d)} \#_{O R} \operatorname{CSPF}^{(d)}\left(v_{j}\right) \tag{E2.8}
\end{align*}
$$

$$
\begin{align*}
& \operatorname{CSPF}^{(d)}\left(c_{i j}\right)=G^{\prime(d)} \#_{A N D} \operatorname{CSPF}^{(d)}\left(v_{j}\right)  \tag{E2.9}\\
& \operatorname{CSPF}^{(d)}\left(c_{i j}\right)=G^{\prime(d)} \#_{\operatorname{NAND} \operatorname{CSPF}^{(d)}\left(v_{j}\right)} \tag{E2.10}
\end{align*}
$$

depending on whether $\mathbf{v}_{\mathbf{j}}$ is a NOR, OR, AND or NAND gate respectively. $\mathrm{F}^{\prime}$ and $\mathrm{G}^{\prime}$ are slightly different from $F$ and $G$ in equations E2.1 through E2.4 and are given in equations E2.11 and E2.12.

$$
\begin{align*}
& F^{\prime}=\cup_{x<i} f\left(c_{x j}\right)  \tag{E2.11}\\
& G^{\prime}=\cap_{x<i, x \neq i} f\left(c_{x j}\right) \tag{E2.12}
\end{align*}
$$

As can be seen, $F$ ' and $G^{\prime}$ depend on how the connections are ordered. For a connection ordered with a smaller ' $i$ ' value, the size of its don't-care set associated with its CSPF is smaller. [MK89] uses some heuristics to order the connections in a circuit and they are listed here.

1) Connections that are connected to input variables are given smaller ' i ' values. This is because such connections are often difficult to remove. In addition, the removal of the other types of connections may cause some gates in the circuit to be removed also and result in a better overall gain.
2) Connections connected to gates with larger fanouts are given smaller ' i ' values than connections connected to gates with smaller fanouts. This increases the chance of removing a gate when all of its output connections are removed.

The computation of the CSPF of a gate is performed exactly as the case of computing the MSPF of a non-reconvergent gate. Fornally,

$$
\begin{equation*}
\operatorname{CSPF}^{(d)}\left(v_{i}\right)=\cap_{1 \leq x \leq k} \operatorname{CSPF}^{(d)}\left(c_{i x}\right) \tag{E2.13}
\end{equation*}
$$

The CSPF of an output gate is the same as its output function. Again, similar to MSPFs, CSPFs are computed from the primary outpits towards to the primary inputs.

The use of E2.13 to compute the CSPFs of every gate is one of the major timesaving factor in using CSPF rather than MSPF as a circuit needs not be evaluated again to obtain the output functions with respect to a reconvergent gate. In addition, as CSPFs are based on a partial ordering of the connections, they need not be recomputed again each time the circuit is transformed.

After calculating the CSPFs or MSPFs of the gates and connections in a circuit, transformations can be applied to reduce their number. Such procedures are explained in the following sections.

### 2.4. Pruning

The pruning procedure removes redundant connections in a circuit. Pruning can either be based on MSPF or CSPF. In order to deteci redundant circuits, the MSPFs or CSPFs of all the connections have to be computed The rules of deciding whether a connection is redundant is as follows :

1) If the gate $v_{j}$ is a NOR or OR gate and $\operatorname{siPF}^{(d)}\left(c_{i j}\right)=0$ or $*$ for all $1 \leq d \leq$ $2^{\mathrm{n}}, \mathrm{c}_{\mathrm{ij}}$ is redundant.
2) If $v_{j}$ is an AND or NAND gate and $\left.\operatorname{SPF}^{\prime}{ }^{d}\right)\left(c_{i j}\right)=1$ or $*$ for all $1 \leq d \leq 2^{n}$, $\mathrm{c}_{\mathrm{ij}}$ is redundant.
3) If $v_{j}$ is a NOT gate and $\operatorname{SPF}^{(d)}\left(c_{i j}\right)=*$ for all $1 \leq d \leq 2^{n}, c_{i j}$ is redundant.

To see why this is true, consider a connection $\mathrm{c}_{\mathrm{ij}}$ connected to an AND gate as shown in Figure 2.5. If $\operatorname{SPF}^{(\mathrm{d})}\left(\mathrm{c}_{\mathrm{ij}}\right)=1$ or $*$ for all $1 \leq \mathrm{d} \leq 2^{\mathrm{n}}$, then $\mathrm{c}_{\mathrm{ij}}$ is actually not needed to turn off the output of $\mathbf{v}_{\mathbf{j}}$ for all combinations of the input variables and still maintains the primary outputs of the circuit. Hence, $c_{i j}$ is redundant and can be removed. This similarly explains the cases for the other gate types of $\mathbf{v}_{\mathbf{j}}$.

The procedure for performing pruning based on MSPFs is given in Procedure 2.4.1.

Procedure 2.4.1-Pruning based on MSPFs.

1) Calculate the output function at every gate.
2) Levelize the circuit with respect to the primary outputs.
3) For every level of gates starting from the one nearest to the primary outputs,

For every gate within a level,
3.1) Compute the MSPF of the gate.
3.2) Compute the MSPF of each of the gate's input connections.
3.3) If a connection is redundant, remove it and possibly the


Figure 2.5. An example of a connection $\varepsilon_{i j}$ to an AND gate.
gates attached to it. Repeat from step 1 until no further improvement can be made.

If CSPFs are used instead of MSPFs in the pruning procedure, Step 3.3 in Procedure 2.4.1 can be modified so that it does not repeat from Step 1. This is given in Procedure 2.4.2.

Procedure 2.4.2 - Pruning based on CSPFs.

1) Calculate the output function at every gate.
2) Levelize the circuit with respect to the primary outputs.
3) For every level of gates starting from the one nearest from the primary outputs,

For every gate within a level
3.1) Compute the CSPF of the gate
3.2) Compute the CSPF of each of the gates input connections.
3.3) If a connection is redundant, remove it.
4) Repeat from Step 1 until no further improvement can be made.

However, the circuit obtained from Procedure 2.4.2 may not always be free of redundant connections. This is because a CSPF does not contain the full don't-care set associated with a connection or a gate. To obtain an irredundant circuit, Procedure 2.4.1 can always be performed after Procedure 2.4.2. This is faster than using Procedure
2.4.1 alone to obtain an irredundant circuit [X90].

### 2.5. Gate Substitution

In gate substitution, a gate in a circuit is selected and the other existing gates are each checked to determine if the latter can replace the former without changing the functions at the primary outputs. This is illustrated in Figure 2.6.

To determine if a gate $\mathbf{v}_{\mathbf{j}}$ can replace another gate $\mathbf{v}_{\mathbf{i}}$, either the MSPF or CSPF of $v_{i}$ can be used. However, CSPF is used in our implementation of the Transduction Method (PTRANS) as the use of MSPF is too time-consuming. In fact, CSPF is used for all of the other transformations described later. The condition for $\mathbf{v}_{\mathbf{j}}$ to be able to replace $v_{i}$ is $f^{(d)}\left(v_{j}\right) \in \operatorname{CSPF}^{(d)}\left(v_{i}\right)$ for all $1 \leq d \leq 2^{n}$. The correctness of this condition follows straight from the definition of the CSPF of $v_{i}$. If $f\left(v_{j}\right)$ is an element of


Figure 2.6. An example of gate substitution.
$\operatorname{CSPF}\left(\mathrm{v}_{\mathrm{i}}\right)$, the functions at the output connections of $\mathrm{v}_{\mathbf{i}}$ can be changed to the function at $\mathbf{v}_{\mathbf{j}}$ without changing the primary outputs. Hence, each of them can be connected to the output of $\mathbf{v}_{j}$ instead of $\mathbf{v}_{\mathbf{i}}$, and $\mathbf{v}_{\mathbf{i}}$ can be removed from the circuit.

The procedure for performing gate substitution is given in Procedure 2.5.1.

Procedure 2.5.1-Gate Substitution.

1) Calculate the CSPFs of all the gates and connections.
2) For every gate $v_{i}$,

For every other gate $\mathbf{v}_{\mathbf{j}}$ which is not a successor of $\mathbf{v}_{\mathbf{i}}$, if $f^{(d)}\left(v_{j}\right) \in \operatorname{CSPF}^{(d)}\left(v_{i}\right)$ for all $1 \leq d \leq 2^{n}$, replace each output connection of $v_{i}$ with the output from $v_{j}$ and remove $v_{j}$ (with the possibility of some other gates in the circuit) from the circuit.
3) Repeat from Step 2 until no further substitution can be performed.
4) Repeat from Step 1 until no further substitution can be performed.

In Procedure 2.5.1, $v_{j}$ must not be a successor of $\mathbf{v}_{\mathbf{i}}$. This is to prevent a loop from being formed during the substitution.

### 2.6. Gate Merging

The gate merging procedure is slightly more complicated than gate substitution as described earlier. The basic idea of this procedure is to select two gates and determines if a third gate can be synthesized with inputs connecing to existing gates in the circuit
other than the two above-mentioned gates so that this third gate can replace them. This results in the saving of a gate and is shown in Figure 2.7.

To perform gate merging, the connectable condition for gates is used. A gate $\mathbf{v}_{\mathrm{i}}$ is said to be connectable to another gate $\mathbf{v}_{\mathbf{j}}$ if the following conditions apply :

1) If $v_{j}$ is a NOR gate and there does not exist a value for $d$ between 1 and $2^{n}$ such $\operatorname{CSPF}^{(d)}\left(v_{j}\right)=1$ and $f^{(d)}\left(v_{i}\right)=1$.
2) If $v_{j}$ is a $O R$ gate and there does not exist a value for $d$ between 1 and $2^{n}$ such $\operatorname{CSPF}^{(\mathrm{d})}\left(\mathrm{v}_{\mathrm{j}}\right)=0$ and $\mathrm{f}^{(\mathrm{d})}\left(\mathrm{v}_{\mathfrak{i}}\right)=1$.
3) If $v_{j}$ is a AND gate and there does not exist a value for $d$ between 1 and $2^{n}$ such $\operatorname{CSPF}^{(d)}\left(v_{j}\right)=1$ and $f^{(d)}\left(v_{i}\right)=0$.
4) If $v_{j}$ is a NOR gate and there does not exist a value for $d$ between 1 and $2^{n}$ such $\operatorname{CSPF}^{(d)}\left(v_{j}\right)=0$ and $f^{(d)}\left(v_{i}\right)=0$.


Figure 2.7. An example of gate merging.

The case of $\mathbf{v}_{\mathbf{j}}$ being an inverter is not listed in any of the conditions as it can be treated as a single-input NAND or NOR gate. In addition to the connectable condition, the intersection operator, $\cap$, on the three-value domair. of $\{0,1, *\}$ is used and is defined in Table 2.5. This operator is symmetric.

The gate merging procedure is given in Procedure 2.6.1. Again, CSPF is used.

Procedure 2.6.1 - Gate merging.

1) Calculate the CSPFs of all the gates.
2) Pick two gates $v_{1}$ and $v_{2}$ such that their CSPFs are intersectable, i.e. the '-' sign in Table 3.1 does not arise.
3) Synthesize another gate $v_{3}$ with $\operatorname{CSPF}$ equals to $\operatorname{CSPF}\left(v_{1}\right) \cap \operatorname{CSPF}\left(v_{2}\right)$. Let $\mathrm{v}_{3}$ be a NOR gate.
4) Search the circuit to obtain the set of gates which are connectable to $v_{3}$. These gates cannot be successors of $v_{1}$ and $v_{2}$. If the set of gates obtained is empty, try $\mathrm{v}_{3}$ being either an OR, AND or NAND gate. If the set is still empty, repeat from Step 2 to try some other pairs of gates.
5) Find the minimal set of connectable gates by going through Steps 5.1 to 5.2.

| $n$ | 0 | 1 | $*$ |
| :--- | :--- | :--- | :--- |
| 0 | 0 | - | 0 |
| 1 | - | 1 | 1 |
| $*$ | 0 | 1 | $*$ |

Table 2.5. Definition of the operator $\cap$.
5.1) For each gate $v_{k}$ in the set, remove it and test if the resulting $f\left(v_{3}\right)$ is still a member of $\operatorname{CSPF}\left(v_{3}\right)$.
5.2) If it is, remove $v_{k}$ from the set.
6) Connect the connectable gates to $v_{3}$. Let $v_{3}$ takes over the output connections of $v_{1}$ and $v_{2}$. Delete $v_{1}$ and $v_{2}$ and possibly some other gates from the circuit.
7) Repeat from step 1 until no further improvements can be made.

Although Step 7 in Procedure 2.6 .1 can repeat from Step 2 instead of Step 1 as CSPF is used, it is found that only very few pairs of gates can be merged in each iteration of Steps 1 through 6 . Hence, it generally saves much more time to repeat from Step 1 after a merge than to continue searching in a highly unsuccessfully search-space by starting at Step 2.

### 2.7. Generalized Gate Substitution

As its name suggests, generalized gate substitution is a more general form of gate substitution. In gate substitution, a gate is checked to see if all of its output connections can be replaced by the output of another gate in the circuit. In generalized gate substitution, each of the gate's output connection is checked if it can be replaced by the output of some other gate instead. Hence, a gate may be substituted by more than one gate. An illustration of this procedure is shown in Figare 2.8 in which $c_{34}$ and $c_{35}$ can be (supposedly) replaced by $c_{14}$ and $c_{45}$ respectively. The crossed-out connections and the gate $v_{3}$ can then be removed from the circuit, resulting in one less gate for the
circuit.

Due to the similarity between generalized gate substitution and gate substitution, the procedure for this transformation is obtained by modifying Procedure 2.5 .1 slightly.

## Procedure 2.7.1 - Generalized Gate Substitution.

1) Calculate the CSPFs of all the gates and connections.
2) For every gate $\mathbf{v}_{\mathbf{i}}$,

For every connection $\mathbf{c}_{\mathbf{i k}}$,
For every other gate $\mathbf{v}_{\mathbf{j}}$ which is not a successor of $\mathbf{v}_{\mathbf{i}}$, if $f^{(d)}\left(v_{j}\right) \in \operatorname{CSPF}^{(d)}\left(c_{i k}\right)$ for all $1 \leq d \leq 2^{n}$, replace $c_{i k}$


Figure 2.8. An example of generalized gate substitution.
with a new connection $c_{\mathbf{j k}}$ if there isn't any connection $c_{\mathbf{j k}}$ originally. Remove $c_{i k}$ with some other gates in the circuit if any.
3) If all of the output connections of $v_{i}$ are aot substituted, undo Step 2.
4) Repeat from Step 2 until no further improvement can be performed.
5) Repeat from Step 1 until no further improvement can be performed.

In this procedure, a gate cannot be partially substituted as this does not result in a better circuit size. Step 3 prevents this from occurring:

### 2.8. Gate Input Reduction

Finally, the fourth transformation available in SYLON-XTRANS is gate input reduction. In this transformation, a new gate $v_{j}$ is synthesized to replace a target gate $v_{i}$ such that the number of inputs of $v_{j}$ is less than that of $\mathbf{v}_{\mathbf{i}}$. After a successful gate input reduction transformation, the total number of connections in the circuit is reduced.

To perform this transformation, a more stringent form of the connectable condition, namely, the effectively connectable condition is needed. A gate $v_{j}$ is said to be effectively connectable to $v_{i}$ if one of the following four conditions is true.

1) If $v_{i}$ is NOR gate, there must be some value of $d$ between 1 and $2^{n}$ inclusive such that $\operatorname{CSPF}^{(d)}\left(v_{i}\right)=0$ and $f^{\prime(d)}\left(v_{j}\right)=1$.
2) If $v_{i}$ is OR gate, there must be some value of $d$ between 1 and $2^{n}$ inclusive such that $\operatorname{CSPF}^{(d)}\left(\mathbf{v}_{\mathbf{i}}\right)=1$ and $f^{(d)}\left(v_{j}\right)=1$.
3) If $v_{i}$ is AND gate, there must be some value of $d$ between 1 and $2^{n}$
inclusive such that $\operatorname{CSPF}^{(d)}\left(v_{i}\right)=0$ and $f^{(d)}\left(v_{j}\right)=0$.
4) If $v_{i}$ is NAND gate, there must be some value of $d$ between 1 and $2^{n}$ inclusive such that $\operatorname{CSPF}^{(d)}\left(v_{i}\right)=1$ and $f^{(d)}\left(v_{j}\right)=0$.

With the effectively connectable condition, the procedure for gate input reduction is as follows :

## Procedure 2.8.1-Gate Input Reduction.

1) Calculate the CSPFs of all the gates in tre circuit.
2) For each gate $v_{i}$,
2.1) Synthesize a new OR gate $v$ which has the same CSPF and function as $\mathbf{v}_{\mathbf{i}}$.
2.2) Search for the set of gates in the circuit which are effectively connectable to $v$. These gates must not be successors of $\mathbf{v}_{\mathrm{i}}$.
2.3) Minimize the number of gates in the set obtained from Step 2.2 by the following :
2.3.1) For each gate $\mathrm{v}_{\mathrm{k}}$ in the set, remove it and test if the resulting $f(v)$ is still a member of $\operatorname{CSPF}(v)$.
2.3.2) If it is, remove $v_{k}$ from the set.
2.4) If the size of the set is less than the number of inputs of $v_{i}$, add a new connection from. each of the gates in the reduced set to the input of $v$ and uspe it to replace $v_{i}$. Otherwise, try synthesizing v as a NOR, AND or NAND gate instead of

NOR.
3) Repeat from Step 2 until there is no further improvement.
4) Repeat from Step 1 until there is no further improvement.

Very frequently, it is found that the law of diminishing returns. applies to Procedures 2.7.1 and 2.8.1. The number of reductions to the circuit that can be made decreases rather rapidly after these procedures are applied for a constant number (once or twice) of times. Hence, the two procedures are combined into one single procedure which is then applied once or twice to a circuit. This combined procedure goes through the circuit and for each gate, it tries to perform generalized gate substitution on that gate. If this is unsuccessful, gate input reduction is then applied (if applicable) to it. The procedures given from Sections 2.4 through 2.8 form the basic tools for the optimization of a multi-level circuit in SYLON-XTR\& NS.

## CHAPTER 3. ;

## PARALLEL IMPLEMENTATION OF SYLON-XTRANS

### 3.1. General Overview

In the parallelization of SYLON-XTRANS, many problems have to be dealt with. This section describes the problems and their solut ons that have led to the present implementation of PTRANS on an Encore 510 Multimax, which is an eight-processor shared-memory multiprocessor.

The first problem concems the size of the input circuit. After several experiments, the synchronization overheads incurred in PTRANS viere found to grow slower than the actual time spent in minimizing the circuit. Hence: the input circuit has to be large enough so that the overheads can be sufficiently masked for achieving good speedups and high efficiencies. However, it is impossible for FTRANS to minimize any arbitrary large circuits as this is bounded by the computer's memory limitation.

To solve this problem, binary decision diagrams (BDDs) are used to represent functions and permissible functions instead of the more traditional SOP form as used in SYLON-XTRANS. BDDs are generally more compact than the SOP representation [B86]. In addition to using BDDs, the file system is also used as a temporary storage. Although the Encore Multimax computer has virtual memory, the amount of swapspace available on our system is limited. Hence, PTRANS has to manage the temporary disk storage explicitly. It selectively stores and retrieves BDDs generated
during program execution to and from the disk.
Unfortunately, some of the circuits (eg. the ISCAS benchmarks) are still too big to be minimized as a whole. Such circuits are partitioned into smaller circuits before minimization and can be merged afterwards. The paritions can either be minimized in parallel consecutively, or in parallel simultaneously. PTRANS performs the necessary intra- and inter-partition load balancing automatically. These modes of parallelism are illustrated in Figure 3.1.

In this chapter, the details of the implementation of PTRANS is given. In Section 3.2, the methods of manipulating BDDs to handle permissible functions are described. Section 3.3 briefly summarizes the partitioning algorithm used to partition large circuits. In Section 3.4, the program model is given followed by descriptions of how the functions and permissible functions of gates in a circuit can be evaluated in parallel. Finally, Sections 3.5 through 3.11 explains the paraliel implementation of the various transformations.

### 3.2. Binary Decision Diagrams

The use of BDDs to represent Boolean functions was formally introduced in [A78] and [B86]. As permissible functions contain a third don't-care value $\left(^{*}\right)$ in addition to the $\{0,1\}$ binary values in ordinary Boolean functions, the original BDD structure has to be modified to represent this additional value [MF89]. Furthermore, PTRANS uses some additional BDD operators which are also described in this section.


Figure 3.1. The three modes of parallelism in PTRANS.
The way in which the don't-care value is represented in a BDD is as follows. Suppose the bit vector $\left(0^{*} 110^{*} 1^{*}\right)$ is to be represented and it corresponds to the truth table shown in Table 3.1. Its BDD equivalent is then given in Figure 3.2. As can be seen, the only modification needed is to have another terminal node in the BDD that
represents the don't-care value.

As can be deduced from Figure 3.2, the size of a BDD is dependent on the ordering of the variables, i.e. the levels at which the input variables appear in the BDD. Some ordering heuristics have been presented in the literature [MWBV88] and [FFK88]. PTRANS uses a heuristic ordering based on the frequency with which each primary input is connected to a gate. The justification is that a primary input that is connected to more gates probably affects more functions, and hence is given a higher priority in the variable ordering. Thus, it is placed nearer to the roots of the BDDs.

| $\mathrm{x}_{1}$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $\mathrm{x}_{2}$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $\mathrm{x}_{3}$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Vect. | 0 | $*$ | 1 | 1 | 0 | $*$ | 1 | $*$ |

Table 3.1. Truth table for the vector $\left(0^{*} 110^{*} 1^{*}\right)$.


Figure 3.2. The BDD for the truth table in Table 4.1.

In PTRANS, there are four new procedures for manipulating these BDDs. They are listed as follows :

1) Test if a function is a member of another function.
2) Test if a function intersects with another function.
3) Test for the connectability condition.
4) Test for the effectively connectablility condition.

These are given in Procedures 3.2.1, 3.2.2, 3.2.3 and 3.2.4 respectively. The relevant fields in the data structure used for the BDDs are basically the same as those described in [B86]. Procedures 3.2.3 and 3.2.4 for testing the connectable and effectively connectable conditions follow straight from their definitions in Sections 2.6 and 2.8 respectively.

Procedure 3.2.1 (BDD1,BDD2) - Tests if BDD1 is a member of BDD2.
/* Input : BDD1 and BDD2.
Output : Returns 1 if BDD1 $\in$ BDD2, 0 otherwise. */

1) If BDD2.val $=$ *, return(1).
2) If BDD2.val $\neq *$ and BDD1.val $\neq$ BDD2.val, return( 0 ).
3) Recursively call on the subtrees of BDD1 and BDD2 to check if the subtrees of BDD1 is a member of their corresponding subtrees of BDD2.

Procedure 3.2.1 is a straight forward implementation of checking if every bit in the vector represented by $\mathrm{BDD1}$ is a subset of the zorresponding bit in the vector represented by BDD2 by traversing both BDDs. When the subset condition fails for a
pair of bit values in the two vectors, the procedure returns a 0 immediately.

Procedure 3.2.2 (BDD1,BDD2) - Tests if two PDDs intersect.
/* Input : BDD1 and BDD2.
Output : Returns 1 if BDD1 $\cap$ BDD2 $\neq \varnothing, 0$ otherwise. */

1) If BDD1.val $=\operatorname{BDD} 2 . v a l \neq{ }^{*}$, return(1).
2) If BDD1.val $=1$ and $\operatorname{BDD} 2 . v a l=0$ or vice-versa, return( 0 ).
3) Recursively call on the subtrees of BDD1 and BDD2 to determine if they intersect.

Similar to Procedure 3.2.1, Procedure 3.2.2 traverses both BDDs to ensure that the corresponding bits in the vectors represented by BīD1 and BDD2 are intersectable. This intersectable condition is violated only when a bit in the first vector is 1 and the corresponding bit in the second vector is 0 or vice-versa. At this point, the procedure stops and returns a 0.

Procedure 3.2.3 (f,SPF,Gate_type) - Tests connectability.
/* Tests if a gate with function f is connectaide to another gate v with CSPF called SPF. Gate_type is the type of gate $v$ is. It can be NOR, OR, AND or NAND.

Input : a function f , a CSPF called SPF and a gate type.
Output : Returns 1 if the connectable condition is true and 0 otherwise. */

1) If SPF.val $=$ *, return(1).
2) If Gate_type $=$ NOR
if SPF.val $=1$ and f.val $=1$, return(0); else if SPF.val $\neq{ }^{*}$, return(1).
3) If Gate_type $=$ OR
if SPF.val $=0$ and f.val $=1$, return( 0 ); else if SPF.val $\neq{ }^{*}$, return(1).
4) If Gate_type $=$ AND
if SPF.val $=1$ and f.val $=0$, return( 0 ); else if SPF.val $\neq{ }^{*}$, return(1).
5) If Gate_type $=$ NAND
if SPF.val $=0$ and f.val $=0$, return(0); else if SPF.val $\neq$ *, return(1).
6) Recursively call on the subtrees of $f$ and SPF to check for effectively connectability.

At each recursion of Procedure 3.2.3, if a pair of terminal values is reached, the procedure checks if the connectable condition defined in Section 2.6 is violated depending on the type of gate $v$ is. Once a violation is detected, the recursion aborts and the procedure returns a 0 . Otherwise, the procedure recursively checks other pairs of terminal values in the two BDDs, $f$ and SPF.

Procedure 3.2 .4 (f,SPF,Gate_type,flag). Tests effectively connectability.
/* Tests if a gate with function f is effectively connectable to another gate v with
CSPF SPF. Gate_type is the type of gate $v$ is. It can be NOR, OR, AND or NAND. 'flag' is an external Boolean variable.

Input : a function $f$, a CSPF called SPF, a gate type, and an external variable flag.
Output : Returns 1 if the effectively connectable condition is true and 0 otherwise.

1) If SPF.val $=$ *, return(1).
2) If Gate_type $=$ NOR
if SPF.val $=1$ and f.val $=1$, return $(0)$; else if SPF.val $=0$ and f.val $=$ 1 , set flag to be true. Otherwise, if SPF.val $\neq{ }^{*}$, return(1).
3) If Gate_type $=O R$
if SPF.val $=0$ and f.val $=1$, return(0); else if $S P F . v a l=1$ and f.val $=$ 1, set flag to be true. Otherwise, if SPF.val $\neq{ }^{*}$, return(1).
4) If Gate_type = AND
if SPF.val $=1$ and f.val $=0$, return(0); else if SPF.val $=0$ and f.val $=$ 0 , set flag to be true. Otherwise, if SPF.val $\neq$ *, return(1).
5) If Gate_type $=$ NAND
if SPF.val $=0$ and f.val $=0$, return( 0 ); else if SPF.val $=0$ and f.val $=$ 1, set flag to be true. Otherwise, if SPF.val $\neq$ *, return(1).
6) Recursively call on the subtrees of $f$ and SPF to check for effectively connectability.
7) The effectively condition is only true if both the procedure returns 1 and flag has been set to true.

Procedure 3.2.4 is very similar to Procedure 3.2.3 except that a Boolean flag is used to record if f is effective with respect to the function SPF, i.e. if f has helped in the setting of any bit of SPF to its value based on the type of gate $v$ is. The remaining conditional statements in the procedure checks for the connectable condition which is already shown in Procedure 3.2.3. Thus, when the procedure returns both a 1 and the
flag has been set, both the effective and connectable conditions are satisfied.

### 3.3. Partitioning Algorithm

In this section, the partitioning algorithm used for breaking up large circuits is briefly described. More details can be found in [DB91].

The partitioning algorithm comprises of seed-clustering and group-migration algorithms. Each execution of the algorithm breaks a circuit into two partitions. The seedclustering algorithm starts by locating two seeds for two partitions chosen such that they are maximally away from all boundary gates like primary inputs and primary outputs in the circuit. They are also as far away as possible from each other.

After the two seeds are located, they are separated into two growing partitions. The other gates not yet considered are placed on a free-list. Considering one partition at a time, a gate is then picked from the list such that the gain obtained by putting it into the partition is maximum. The cost function for calculating the gain will be described later.

When the free-list becomes empty, Kernighan-Lin's algorithm [KL70] is then used to swap pairs of gates between the partitions. The pairs of gates are selected such that swapping them result in more gain in the overall qualities of the partitions.

The cost function used to measure the amount ci gain of a gate with respect to a partition is an estimate of the size of the don't-care set associated with the gate. This can be found by choosing random vectors to simulate the circuit the gate is in. From
the frequency of 0 's and 1 's appearing in each cornection, the don't-care set can be estimated. More details of this can be found in [DB91].

With this partitioning algorithm, large circuits can be partitioned and optimized in parallel. The details of this parallel implementation is given in the following sections.

### 3.4. Program Model

As mentioned earlier, large circuits have to be partitioned before they can be minimized. At the implementation level, no distinction is made between a partition of a circuit and a whole circuit. PTRANS can be fed with as many partitions as possible simultaneously under the constraint caused by the amount of memory available. There is no relation between the number of input partitions and the number of processors PTRANS uses.

PTRANS uses a multiple master-slave model. This is very similar to the normal master-slave program model, whereby the master distibutes computations to the slave processes and is also in charge of synchronizing thern. The results of the computations are then passed back to the master. The only differences between the model PTRANS uses and the normal master-slave model are that multiple masters are present in PTRANS, and each slave does not always belong tr. the same master. In PTRANS, each master or slave is actually a process in the system. A processor is assumed to be always allocated to a process by the operating system. The number of processes can vary from one to the number of processors available on the system.

At any instant of time, only one master is associated with a partition. This master is responsible for the whole minimization process of the partition. During the minimization of its partition, the master will never, be used for the minimization of other partitions. As for the slaves, they stay in a shared slave pool. Whenever a master reaches a point during its execution where it can distribute its load to other processes, it will enter exclusively into the slave pool and try to get as many slaves as possible from the pool. It then distributes the load to those slaves When these slaves have finished their computations, they return to the slave pool awaiting for future masters. Whenever a partition has been minimized, the corresponding master becomes a slave and it too enters the slave pool.

In order to efficiently utilize the processors, each master cannot own slaves throughout the whole minimization process of a partition as this will deny the other masters of slaves. In PTRANS, there are several entry and exit points. Entry points are locations where slaves can join a master in the minimization of a partition. Similarly, exit points are locations where slaves can leave a master and return to the slave pool. After a slave has been sought for help by a master, it will enter at an entry point determined by the master, perform the computations in parallel with the master and other slaves, exit at the next exit point and return to the slave pool. Between every pair of entry and exit points is a well-defined piece of job such as gate substitution etc. An illustration of this master-slave relation is shown in Figure 3.3. Using this slave pool, the load can be distributed to idle processors. This forms the basis of the load balancing between the processors in PTRANS.


Figure 3.3. A sample timing diagram of two masters and two slaves.
Beside using the master-slave model, PTRANS also use a semi-distributed memory model. Every process has some semi-private memory locations pre-allocated to it. This is to avoid the contention in allocating memory for frequently used data structures such as BDDs since allocating shared-memory is a sequential bottleneck. This set of memory locations is classified as private memory because only the owning process can allocate memory out of its set. However, it is semi-private as data structures allocated from a set can be read and de-allocated by other processes.

With this basic model, PTRANS is able to minimize multiple partitions simultaneously. Let $p$ be the number of partitions and $P$ be the number of processors. Initially, there are $\min (p, P)$ masters. This number will reduce gradually. If $P$ is greater than p, there will be P-p initial slaves also. Each mas er is allocated a list consisting of
$\mathrm{p} / \mathrm{P}$ partitions. Since P does not generally divide p , some masters may have one partition more than the other masters. These partitions can be minimized in parallel without any dependency between them. Whenever a master has processed all of its allocated partitions, it performs a scan of the other masters' lists of partitions and looks for the first uncomputed partition. It then removes this partition from the list and minimizes it. If the master cannot find such a partition, it checks if it is the last master among all of the P processes. If so, this master will send a termination message to each of the other $\mathrm{P}-1$ slaves and all of the P processes will then exit, thus terminating the whole program. Otherwise, this master will change its status to a slave and enters the slave pool.

In addition to this high level inter-partition parallelism, PTRANS is also able to apply the Transduction Methods on a partition in parallel. This are described in the coming sections.

### 3.5. Discussion of Number of Partitions

An obvious way of extracting significant speadup out of the logic synthesis application is to generate a large number of partitions and synthesizing each partition independently. The results of the individual partitions are then merged back. Unfortunately, such an approach has the problem that with increasing number of partitions, the quality of the overall circuit degrades. This is because each synthesis procedure of a partition only synthesizes within tre partition by treating it as an independent block. It does not take any global information into consideration during
minimization. A good partitioning algorithm that can guarantee minimum degradation in the circuit quality is desirable. Examples of existing partitioning algorithm are BEAT-NP [CHNS88], COROLLA [DBK90], and that of Banerjee [DB91].

In the interest of better quality, one should thererore choose a minimum number of partitions. Then, one is forced to resort to intra-partition parallelism which is much harder to exploit. One may not get good speedup within a partition. There is clearly a tradeoff between result quality and runtime determined by an optimal number of partitions. Such a theory needs to be developed but is outside the scope of this thesis.

### 3.6. Parallel Evaluation of Functions and CSPFs of Gates

To exploit intra-partition parallelism, the parallel evaluation of functions and CSPFs of gates is discussed in this section. The evaluation of MSPFs is slightly different from that of CSPFs and is deferred to the next section.

The parallel evaluation of the output functions of gates is similar to the parallel methods of logic simulation [SB88] and circuit partit.on approaches to fault simulation [PBP91]. From the definition of a level in a circuit $n$ Section 2.1, it can be seen that gates within the same level with respect to the primary inputs can have their functions evaluated in parallel. Similarly, the CSPFs of gates having the same level number with respect to the primary outputs can be computed in jarallel too. This is illustrated in Figures 3.4(a) and (b).


Figure 3.4(a). An example of parallel evaluation.


Figure 3.4(b). A sample timing diagrari for Figure 3.4(a).
To traverse the circuit so that gates in the same level can have their functions evaluated in parallel, every process (both master aril slaves) working on the circuit needs an input and an output queue. Initially, the primary inputs of the circuit are evenly distributed among the input queues of these processes. There is a counter
associated with each gate which is initialized to zero. Whenever a process takes a gate v from its input queue and evaluates its function, it increments the counters in each of the immediate successors of $v$. If the counter in a gate equals to the number of its input connections (signifying that all its inputs have been processed and hence the output function of the gate should be evaluated), this counder is reset to zero and the gate is enqueued into the output queue of the process. After a process has processed all of its input queue, it examines the input queues of the other processes, picks the longest queue, removes half of its contents, puts those into its own input queue and continues processing the queue. When all of the input queues have been emptied, a level of gates have been processed. The master of the circuit then concatenates all of the output queues into a single queue and distributes the gates in this queue evenly among the input queues. After this, all processes involved in th:s circuit will continue processing their input queues as described earlier.

Whenever the master finds that all of the output queues are empty after a level has been processed, the functions at all of the gates in the circuit have been evaluated. The slaves will then return to the slave pool.

The evaluation of CSPFs is similarly computed, zxcept that the circuit is levelized with respect to the primary outputs and traversed backwards.

For small circuits, the CSPFs can be stored in the main memory after they are evaluated. However, for more complicated circuits, there is insufficient memory to hold all of these permissible functions simultaneously. To avoid this problem, some of the

CSPFs are transferred to the disk. In order to minirnize the number of disk accesses when evaluating such functions, the permissible function of a gate is not stored into the disk immediately after it is evaluated. In fact, it will be held in the main memory until the CSPFs of all of the input connections of the gate 'have been computed. After this, it is packed into a contiguous format and sent to the disk.

As for the functions of the gates, they generaly require much less memory for storage than CSPFs. This is because the functions ay connections are the same as that of the gate they are connected to, whereas thei: CSPFs are different from that of the gate. Hence, such functions are not stored in the disk.

Another slight difference between the evaluation of functions and CSPFs is the granularity in which these two are performed. For normal functions, the evaluation of the whole circuit is treated as a single task. Thus, the master will only enter the slave pool at the beginning of this task to look for slaves. On the other hand, the evaluation of CSPFs is more time-consuming as the BDDs needed to represent these functions are generally larger. Additional processing is also needed to pack these BDDs for disk storage. Hence, the evaluation of permissible functions is broken up into a smaller grainsize. This grainsize is set at the levels of the circuit. At this grainsize, the evaluation of the gates at the same level is treated as $:$ task and the master is allowed to enter the slave pool to obtain slaves for each level of ne circuit.

It should be noted that the above approach is one way of exploiting the parallelism in the evaluation of functions and CSPFs. Another way would be to partition the input
space on different processors and letting each processor to perform function and CSPF evaluations on its input vectors. For example, with two processors, Processor 1 might be processing the first $\mathrm{d} / 2$ bits of the vectors while Processor 2 is in charge of the remaining bits. Although this is conceptually simple to parallelize in the SOP representation, the difficulty comes when using BDDs. With the splitting of the input space, multiple BDDs are needed to represent a single function. The amount of subtree-sharing in these BDDs will thus be smaller as compared to that in a single BDD representing the same function if input space haó not been splitted. This would increase the amount of memory needed by the BDDs.

### 3.7. Parallel Pruning

The pruning procedure can be broken down ints" two parts. The first part consists of identifying the redundant connections and the seccnd performs the actual removal of the redundant connections.

Pruning based on CSPF is slightly different from pruning based on MSPF. This is because the removal of a redundant connection does not invalidate the CSPFs of other gates and connections whereas this is not true with MSPF. Therefore, for pruning based on CSPF, the CSPFs of all the gates and connections can be first generated before performing any redundancy removal. As the CSPFs are generated, the connection wires are checked to see if the wires can be pruned. If so, such connections are marked. The generation and checking of the CSPFs can be performed in parallel as described in the previous section. This is illustrated in Figure 3.5.


Connections $c_{1}, c_{2}, c_{3}$, and $c_{4}$ are checked for redundancy simultaneously. $c_{1}$ and $c_{2}$ are found to be redundant and are marked in parallel but are removed sequentially

Figure 3.5. An example of parailel pruning.
After all of the connections have been checked for redundancy, the master process then goes through the marked connections and remcive them sequentially. This is not performed in parallel as the time taken to adjust a few pointers during the removal of a redundant connection is negligible as compared to the time needed to detect its existence. In addition, the number of redundant connections is usually very small as compared to the total number of connections in the circuit.

As.for pruning based on MSPF, both the computation of MSPFs and the removal of redundant connections have to combined into a single phase to avoid redundant work. This is because after a connection is found to be redundant and pruned, the MSPFs of all other connections and gates are invalir!ated and have to be recomputed. The grainsize of the computation of the MSPFs is set to the level of a circuit and is similar to the case with CSPFs. Each process computes the MSPF of a gate or a connection as described in Section 2.2 and checks for redundancy of a connection after
its MSPF is computed. If a connection is found to be redundant, it is recorded in a shared variable readable by every process working, on the same circuit. During the computation of the MSPFs, every process checks this variable periodically to determine if a redundant connection has been detected. When this is set to true, the slaves will then return to the slave pool whereas the master process will perform the removal of the redundant connection. After this, it restarts the compcration of the functions and MSPFs of the circuit. This cycle is repeated until no further redundant connection can be found.

As with CSPF, there is insufficient memory to are the MSPF of every gate and connection. However, MSPFs are not stored in the disk since they are not needed for any other transformation. The MSPF of a gate or connection is deleted once it has been used by all of the relevant immediate predecessor gates or input connections. In the pruning procedure of PTRANS, pruning with CSPF is first executed before pruning with MSPF. This combination yields an irredundant circuit in a shorter time then pruning with MSPF alone.

### 3.8. Parallel Gate Substitution

The main idea of gate substitution is to search ti:e given circuit for a pair of gates such that one gate (the candidate gate) can replace the other (the replaced gate). As there are possibly many pairs of gates satisfying the gate substitution condition at a time, the gates are ordered and searched so that the reslaced/candidate gate is as near to the primary outputs/inputs as possible. This is to minimize the occurrence of a pair of candidate and replaced gates such that the candidate gate is a successor of the replaced
gate. In this case, it is impossible to perform the sutstitution since the resulting circuit will not be loop-free.

To search for such pairs of gates in parallel, the gates of the circuit are arranged in two shared queues. The first queue, Q1, contains the gates traversed in a breadth-firstsearch from the primary outputs to the primary inputs. The second queue, Q 2 , contains the same gates arranged in reversed order, i.e. from primary inputs to primary outputs. This is illustrated in Figures 3.6(a) and (b).

To search for a pair of gates for substitution, a process first exclusively dequeues a gate $v$ from Q 1 . It then scans the gates in Q 2 from hedad to tail and stops when it finds a gate which can substitute for gate $\mathbf{v}$. Next, the process records this pair of gates and informs the other processes working on the same circuit to stop their search by setting a shared flag that is periodically monitored by them.

When a process finds that the flag has been set, it will return to the slave pool if it is a slave. Otherwise, if it is the master, it waits until all of its slave processes have returned to the pool and then performs the substitution. It then goes into the slave pool again to get more slaves and continues searchirg for other pairs of gates for substitution.

In order to provide a better load balancing, a mester will also try to obtain slaves from the slave pool during the search if it has not yet obtained $\mathrm{P}-1$ slaves, where P is the number of processors PTRANS is executing on. To do this, each time the master has exclusively dequeued a gate from Q 1 , it will enter the slave pool to look for idle


Figure 3.6(a). Searching for substitutes in parallel.


Figure 3.6(b). A sample timing diagran for Figure 3.6(a).
processors. These new slaves will then help in the search by entering the substitution procedure and each exclusively dequeues a gate from Q1. Every slave is blind to the presence of other slaves.

When Q1 becomes empty, this marks the ead of an iteration in the gate substitution procedure. All slaves will return to the sl:ve pool. If some substitution has
been performed before Q1 becomes empty, the master will set the queues up for the next iteration and initiate the re-evaluation of the functions and permissible functions of the circuit . If not, it will proceed to the next transformation procedure.

### 3.9. Parallel Gate Merging

Gate merging can either use CSPFs or MSPFs. However, in PTRANS, CSPFs are used since the computation of MSPFs is a time-consuning process.

Although CSPFs allow multiple pairs of gates to be merged before re-evaluating, this approach is not used since the number of possible merges with each evaluation of the CSPFs is very small (usually less than 3). Hence, instead of wasting processing time to look for another pair of gates to be merged after a pair has been found, it is more worthwhile to recompute the CSPFs of the gates anid connections and start the search over again.

To look for a pair of gates to merge, gates nearest to the primary outputs are examined first. This is because in gate merging, a thitd gate needs to be synthesized to replace a pair of gates. However, the immediate predecessors of this gate cannot be successors of the gates to be replaced so as to maintain a loop-free circuit. As gates nearest to the primary outputs have fewer successois, this ordering creates a higher probability of being able to synthesize the third gate.

The data structure to iterate the search space for mate merging involves only a single shared queue which contains all of the gates in the circuit traversed in a breadth-
first-search starting from the primary outputs. This queue comes with a 'fetch-andadvance' operator. This operator is a variant of tie fetch-and-add primitive and it atomically returns a copy of a pointer pointing to a gate in the queue and advances the pointer to the next node in the queue. This pointer criginally points to the head of the queue. The queue being operated on by this operator remains intact.

The main loop of the gate merging procedure requires every process to 'fetch-and-advance' for a gate from the single queue. Let $\bar{j}$ be the gate which is returned by the fetch-and-advance operator. After obtaining v, a process then scans the same queue from the its head to tail to look for a gate other than v whose CSPF intersects with the CSPF of $v$. This is illustrated in Figure 3.7(a) and (b). Let $F$ be the intersection of these two CSPFs. After this, it tries to synthesize a third gate, $y$, to substitute the pair of gates. To synthesize $y$, the process again scans the gates in the queue, picks those gates that are connectable to v and adds them to the immediate predecessor set of y . After this, it checks if the resulting function at $y$ is a member of $F$. If so, this process then minimizes the set of connectable gates obtained and informs all of the other processes working on the same circuit to stop by means of setting a shared flag that is being constantly polled. The stopped slaves then return to the slave pool and the merging is performed by the master after every slave has returned. Following the merge, the master re-initiates the gate merging procedure for the netwerk until no further merge can be found.

Gate merging usually takes much longer time as compared with the other transformations. Thus, the grainsize of this procedure has to be small enough to ensure


Gates E and F are fetch-and-advanced by Processes 1 and 2
Figure 3.7(a). Parallel searching in gate merging.


Figure 3.7(b). A sample timing diagrara for Figure 3.7(a).
a good load balance. If this is not so, the following scenario might occur. Suppose there are 2 processes (on 2 processors) minimizing 2 partitions of a circuit simultaneously. At time $t=10$, Process 1 may be working on Partition 1, when Process 2 looks for a slave for gate merging. Process 2 cannot find any slaves and has to work alone. At time $t=$ 12, Process 1 becomes idle after having minimizing its partition. As gate merging may take a long time, Process 2 does not finish until $t=20$. Hence, a processor is idle from $t$
$=12$ to $\mathrm{t}=20$ and this is highly inefficient.

After several experiments, the following proctdure to choose the grainsize was adopted. The program is written such that every time after the master has 'fetch-andadvanced' for a gate, it enters the slave pool to look for more slaves unless it already owns P-1 slaves. Each of these additional slaves th:n proceed straight to 'fetch-andadvance' for their gates. This is transparent to the other processes already working on the partition. At this granularity, the processors are more efficiently utilized.

### 3.10. Parallel Generalized Gate Substitution/Gate Input Reduction

Generalized gate substitution and gate input rediction generally do not reduce the size of circuit as much as the previous transformations in relation to the amount of processing time they take. Therefore, these procedures are combined into a single procedure and is applied to the circuit only a constant number of times while the former transformations iterate until no further improvement.can be made to the circuit.

In the combined procedure, a gate is first examined to see if it can be generallysubstituted by other gates. If not, gate input reduction is then applied. In gate input reduction, a new gate is synthesized for an existing ate such that the gate has smaller number of inputs than the original gate. This, new gate can either be a NOR, OR, AND or NAND gate. NOT gates are equivalent to single-input NAND or NOR gates.

To perform parallel generalized gate substitutio-/gate input reduction, two queues are used. In the first queue, Q1, gates are arranged in a breadth-first-traversal order from the primary inputs towards the primary outputs as usual. On the other hand, the contents of the second queue is different. This queue contains four transformation events for each gate in the circuit. In each event, there is a label field and a gate field. These four label fields for a gate are marked AND, OR, NOR, NAND respectively. The use of this field will be obvious later. The gate field contains a pointer to the corresponding gate.

The second queue, Q 2 , is divided into two se tions . In the first section, all the events marked 'AND' are linked consecutively and the events for the gates nearer to the primary outputs are placed nearer to head of the queue. In the second section, the other remaining events are linked up such that the events for the same gate are grouped consecutively and the sequencing of the group of events for each gate is same as the sequencing of the gates in the first section. An exampee of this arrangement is shown in Figure 3.8.

In an iteration of the transformation, each process exclusively dequeues an event from the second queue. The master process will also :ook for more slaves in the pool at this time unless it already owns P-1 slaves. These new slaves will immediately exclusively dequeues an event each. If the event is marked 'AND', the process first tries to perform generalized gate substitution on the gate in the event. The candidate gates for the substitution is obtained by scanning the gates in the first queue in the order in which they are enqueued. If a successful substitution is found, the other processes


Figure 3.8. An example of the queues in geraralized gate substitution and gate input reduction.
will be halted and the slaves will return to the slave fool. The master then performs the transformation and gets more slaves which will continue to dequeue events from the second queue.

On the other hand, if the substitution is not successful, the process tries to perform gate input reduction on the gate in the dequeued evènt. Similarly, all other processes will be halted if a reduction is found to be successful. Again, the physical reduction is done by the master, which first ensures that all of its slaves have returned to the pool. During the reduction, the type of the gate to be syrihesized is the same as the label field in the event. For example, if the event is marked 'AND', the synthesized gate will be an AND gate. After the transformation, slaves will be again employed to consume
the events in the second queue until it is empty.

For the other events which are marked NOR, OR, and NAND, the process in charge of such an event only tries to perform gate input reduction. Therefore, the arrangement of the events by first having a section of 'AND' events ensures that generalized gate substitution is tested before gate input reduction is considered for a gate.

During the physical transformation of a circuit, some gates may be deleted and some of the events become invalidated. Deleted gates at this stage are only marked 'deleted'. Whenever a process extracts an event with a deleted gate, it will ignore that event and continues with the next one. The end of the procedure is reached when the event queue becomes empty. At this point, the master cleans the circuit up by freeing those gates marked 'deleted'.

### 3.11. Ordering of Search-Spaces

The arrangement of the gates in the queues described in the various transformations is performed so as to avoid huge differences in the search-spaces when the program is executed on the same set of data with different number of processes. It tries to force the processes to look at the gates in a syecified order so that the resulting qualities of the circuit will not differ significantly.

However, a process may sometimes be faster that others due to variances in system load. This sometimes results in super-linear speedups and slightly different
circuit qualities as the sequence of gates being transformed varies from one execution to another.

To avoid this problem, priorities can be assignec, to order the gates in these queues which impose a stricter ordering in the examination of the search-space. However, this is not implemented in PTRANS as the ordering employed in PTRANS are after all heuristics that do not guarantee optimum results. Hence, some degree of randomness in iterating the search-space could even be beneficial when circuit quality is concerned although this could give super-linear speedups. This is evident in the non-degrading circuit qualities over different number processors as will be presented in the next chapter.

## CHAPTER 4.

## EXPERIMENTAL RESULTS

### 4.1. Overview of Experiments

In this chapter, the experimental results from PTRANS is presented. The initial networks are obtained by using MIS 2.1 [BRSW87i] to map the MCNC and ISCAS benchmarks into simple gates. The ISCAS benchmarks are partitioned into smaller circuits. These circuits and partitions are subjected to the following sequence of transformations, the order of which can be changed with ease.

1) Pruning with CSPF.
2) Gate substitution.
3) Pruning with CSPF.
4) Pruning with MSPF.
5) Generalized gate substitution/gate input reduction.
6) Gate merging.

Very frequently, the initial networks produced by MIS 2.1 are found to have only very few redundant connections. Consequently, pruring with MSPF is not used after Step 1. However, after gate substitution is performed. it is usual to find more redundant connections. Thus, pruning with both CSPF and MSFF are applied in Steps 3 and 4 to remove all redundant connections. Gate merging is aoplied last as it takes the longest time for execution and thus saves more time whea applied to circuits after being
minimized by the other transformations.

### 4.2. Circuit Degradation with Number of Processors

In this section, the relation between the final circuit quality and the number of processors used is investigated. The results are tabulated in Table 4.1 where $g$ refers to gate count and c refers to number of connections.

When a circuit has to be broken down to multiple partitions, each partition is minimized one after another. The qualities obtained by minimizing the partitions simultaneously is identical to those obtained by consecutive minimization.

On the whole, the circuit qualities do not degrade with increasing number of processors. In fact, some of them even show better circuit qualities. These slight variances in qualities are due to the fact that when cifferent number of processors are

| Circuit | No. of Partitions | $\begin{aligned} & \hline \text { Initial } \\ & (\mathrm{g} / \mathrm{c}) \\ & \hline \end{aligned}$ | $\begin{gathered} \hline 1 \text { processor } \\ (\mathrm{g} / \mathrm{c}) \end{gathered}$ | $\begin{gathered} 2 \text { processors } \\ (\mathrm{g} / \mathrm{c}) \end{gathered}$ | $\begin{gathered} 4 \text { processors } \\ (\mathrm{g} / \mathrm{c}) \end{gathered}$ | $\begin{gathered} 8 \text { processors } \\ (\mathrm{g} / \mathrm{c}) \end{gathered}$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| f51m | 1 | 131/270 | 81/157 | 81/157 | 81/157 | 81/157 |
| 5xpl | 1 | 129/279 | 79/153 | 79/153 | 78/151 | 75/147 |
| 9 sym | 1 | 205/470 | 173/360 | 173/362 | 180/379 | 187/394 |
| bw | 1 | 205/481 | 145/289 | 146/271 | 150/298 | 149/300 |
| sao2 | 1 | 129/310 | 99/213 | 99/2:3 | 109/231 | 111/235 |
| vg2 | 1 | 158/391 | 73/163 | 73/162 | 73/162 | 73/162 |
| rd73 | 1 | 135/325 | 115/243 | 115/243 | 115/241 | 110/235 |
| duke2 | 1 | 485/1224 | 352/736 | 344/715 | 343/718 | 352/737 |
| alupla | 1 | 114/223 | 103/205 | 103/2.05 | 103/205 | 103/205 |
| misex1 | 1 | 69/154 | 51/102 | 51/102 | 50/100 | 51/101 |
| misex 2 | 1 | 87/233 | 82/178 | 82/178 | 87/178 | 82/178 |
| misex3c | 1 | 493/1231 | 363/776 | 360/770 | 355/755 | 356/766 |
| C432 | 2 | $198 / 411$ | 166/345 | 166/345 | 166/345 | 166/345 |
| C499 | 2 | 526/942 | 493/892 | 493/892 | 493/892 | 493/892 |
| C880 | 4 | 342/688 | 313/362 | 313/352 | 313/362 | 313/362 |
| C1355 | 4 | 492/1018 | 507/1022 | 507/1G22 | 507/1022 | 507/1022 |
| C1908 | 4 | 599/1220 | 448/914 | 449/914 | 448/914 | 453/940 |

Table 4.1. Circuit quality versus the number of processors used.
used, different number of gates are tested for the possibility of transformation at the same time. As the timings of multiple processes are indeterminate, this may result in different sequences of gates being transformed which affects the final quality of the circuit.

### 4.3. Efficiency of Intra-Partition Load Balancing

After studying the effects of multiple processors on the circuit quality, this section reports on the efficiency of the implementation of PTRANS for a single circuit or partition, which is based on the speedups obtained and load balance of the processes.

The speedups in Table 4.2 is computed using the longest processing time taken among the processes rather than user time as the disk used as a temporary storage is a sequential bottleneck. This can be avoided by using multiple disks.

| Circuit | 1 processor <br> Time <br> (sec) | 1 processor | 2 processors | 4 processors | 8 processors |
| :---: | :---: | :---: | :---: | :---: | :---: |
| f51m | 217 | 1.0 | 1.8 | 3.2 | 3.4 |
| 5xp1 | 182 | 1.0 | 1.9 | 2.9 | 4.4 |
| 9 sym | 5408 | 1.0 | 2.8 | 4.1 | 8.3 |
| bw | 793 | 1.0 | 2.2 | 2.8 | 3.7 |
| sao2 | 1556 | 1.0 | 1.9 | 3.6 | 3.5 |
| vg2 | 1603 | 1.0 | 1.6 | 2.8 | 4.6 |
| rd73 | 967 | 1.0 | 2.1 | 3.0 | 5.5 |
| duke2 | 19650 | 1.0 | 2.5 | 4.2 | 7.3 |
| alupla | 6560 | 1.0 | 1.9 | 3.6 | 6.5 |
| misex1 | 115 | 1.0 | 1.8 | 3.2 | 4.8 |
| misex2 | 264 | 1.0 | 1.7 | 3.1 | 3.7 |
| misex3c | 72834 | 1.0 | 2.5 | 3.7 | 4.9 |

Table 4.2. Speedups for circuits using intra-partition parallelism on one partition.

Table 4.2 shows cases of super-linear speedups for some circuits like 9 sym and duke2 as the number of processors used varies. This is again due to the varying sequence of gates being transformed.

On the other hand, PTRANS produces consistent final qualities for the circuit alupla when minimized by $1,2,4$ and 8 processors as shown in Table 4.1. It is very likely that the sequences of gates being transformed are the same throughout these runs. The speedups obtained for minimizing this circuit are graphed in Figure 4.1. The deviation from the ideal linear speedup is small.

As the sequence of gates being transformed varies from one execution to another, the speedups shown in Table 4.2 is not sufficient to show the efficiency of PTRANS. Table 4.3 shows the actual load balance when 8 processes are used. The values are obtained by measuring the processing time of each of the process in each run, and the longest of which is scaled to 100 time units. The rest of the processing times are expressed as a percentage of this time. As shown in '.able 4.3, the processes' loads fall


Figure 4.1. Speedup obtained for minimizing the circuit alupla.

|  | Proc. <br> 1's <br> load <br> $(\%)$ | Proc. <br> 2's <br> load <br> $(\%)$ | Proc. <br> 3's <br> load <br> $(\%)$ | Proc. <br> 4's <br> load <br> $(\%)$ | Proc. <br> $5 ' s$ <br> load <br> $(\%)$ | Proc. <br> 6's <br> load <br> $(\%)$ | Proc. <br> 7's <br> load <br> $(\%)$ | Proc. <br> 8 's <br> load <br> $(\%)$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| f51m | 100.0 | 93.7 | 77.7 | 84.1 | 65.1 | 65.1 | 60.3 | 60.3 |
| 5xp1 | 100.0 | 97.6 | 87.8 | 87.8 | 80.5 | 80.5 | 80.5 | 75.6 |
| 9sym | 100.0 | 91.2 | 88.6 | 87.3 | 84.7 | 86.7 | 81.0 | 80.6 |
| bw | 100.0 | 59.0 | 71.0 | 82.0 | 63.1 | 47.5 | 47.5 | 50.2 |
| sao2 | 100.0 | 91.8 | 82.4 | 82.4 | 82.3 | 78.4 | 84.4 | 85.0 |
| vg2 | 100.0 | 92.8 | 87.5 | 87.5 | 83.8 | 83.2 | 82.9 | 85.8 |
| rd73 | 100.0 | 96.0 | 96.0 | 96.0 | 91.0 | 84.2 | 83.6 | 81.9 |
| duke2 | 100.0 | 99.9 | 94.1 | 88.3 | 83.5 | 87.1 | 84.3 | 86.5 |
| alupla | 100.0 | 90.7 | 93.7 | 87.1 | 87.5 | 84.0 | 87.1 | 82.2 |
| misex1 | 100.0 | 91.7 | 87.7 | 83.3 | 83.3 | 75.0 | 66.7 | 62.5 |
| misex2 | 100.0 | 68.1 | 65.3 | 65.3 | 72.2 | 59.7 | 59.7 | 62.5 |
| misex3c | 100.0 | 96.5 | 94.6 | 95.6 | $95 .$. | 94.0 | 94.1 | 94.5 |

Table 4.3. Load balance for circuits using intra-partition parallelism on one partition on 3 processors.
above $80 \%$ of the largest load in most cases, which implies that the efficiency is greater than 0.8 most of the time.

### 4.4. Efficiency of Inter-Partition Load Balancing

In Section 4.3, the efficiency of PTRANS in minimizing a single partition or circuit is examined. As large circuits needs to be broken into multiple partitions before it can be minimized, this section investigates the' efficiency of PTRANS when minimizing these partitions simultaneously.

By Amdahl's Law, the efficiency of any farallel programs decreases with increasing number of processors. When multiple partitions are minimized simultaneously, the average number of processors per partition is smaller than when a single partition is minimized by the same number" of processors. Hence, a higher efficiency is expected.

| Circuit | No. of <br> partitions | 1 processor <br> Time <br> (sec) | 1 <br> processor | 2 <br> processors | 4 <br> processors | 8 <br> processors |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| C432 | 2 | 4911 | 1.0 | 2.0 | 3.4 | 5.9 |
| C499 | 2 | 5231 | 1.0 | 1.9 | 3.7 | 7.5 |
| C880 | 4 | 1535 | 1.0 | 2.0 | 3.8 | 7.0 |
| C1355 | 4 | 33517 | 1.0 | 20 | 3.9 | 7.4 |
| C1908 | 4 | 6219 | 1.0 | 2.0 | 3.8 | 5.3 |

Table 4.4. Speedup for multiple partitions of circuits minimized simultaneously with combination of inter- and intra-partition parallelism.

From Table 4.1, the final qualities of the ISCAS circuits except C1908 remains consistent. Hence, it is also very probable that the sequences of transformations during the minimization of these circuits are the same throughout the runs from 1 to 8 processors. The speedups obtained for such circuits are shown in Table 4.4. These speedups are much nearer to linear than those shown in Table 4.2, suggesting the high efficiencies achieved. This efficiency is also expressed in terms of the load balance between the processes in Table 4.5.

In Table 4.5, again, the processing time of each process is expressed as a percentage of the longest processing time in each rur. As can be seen, the load of each

| Circuit | Proc. 1's load (\%) | Proc. 2's load (\%) | Proc. 3's load (\%) | Proc. 4's load (\%) | Proc. 5's load (\%) | Proc. <br> 6's <br> load <br> (\%) | Proc. 7's load (\%) | Proc 8's load (\%) |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| C432 | 100.0 | 98.9 | 81.3 | 80.7 | 77.1 | 80.7 | 81.1 | 73.9 |
| C499 | 100.0 | 99.0 | 94.3 | 99.0 | 98.4 | 96.1 | 98.8 | 96. |
| C880 | 100.0 | 94.0 | 98.6 | 96.2 | 96.2 | 97. | 95.8 | 8.6 |
| C1355 | 100.0 | 92.4 | 97.6 | 95.3 | 92.2. | 95.8 | 94.2 | 91.2 |
| C1908 | 100.0 | 95.5 | 96.8 | 97.6 | 95.2 | 95.0 | 95.8 | 94 |

Table 4.5. Load balance for 8 processors running on multiple partitions of the circuits simultaneously with combination of interand intra-partition paralleism.
process is greater than $90 \%$ of the most heavy load for every circuit except for $C 432$. Even for C432, the largest load imbalance is a mere $26 \%$. This shows the effectiveness of the dynamic load balancing strategy used.

Finally, a comparison is made to investigate the differences between using interpartition load balancing and not using it. To do this, every partition of each circuit is minimized one after another using 8 processors. The longest processing time each partition takes is recorded and is shown in Columns 3 through 6 of Table 4.6. Column 7 shows the sum of these times for each partition. Hence, if the partitions of each circuit are minimized in parallel consecutively, the tetal time needed for each circuit is limited by the longest processing time for each partition of the circuit and this value is reflected in Column 7. In Column 8, the longest processing times taken when all of the partitions of a circuit are minimized concurrently are recorded.

From the table, it is important to note that there is a large disparity between the times in Column 3 through 6 for a circuit. For example, this varies from 24 seconds to 3375 seconds for C1355.

| Col. 1 | Col. 2 <br> No. of <br> partit- <br> ions | Col. 3 <br> Longest <br> time for <br> Part. 1 <br> $(\mathrm{sec})$ | Col. 4 <br> Longest <br> time for <br> Part. 2 <br> $(\mathrm{sec})$ | Col. 5 <br> Longest <br> time for <br> Part. 3 <br> (sec) | Col. 6 <br> Longest <br> time for <br> Part. 4 <br> $(\mathrm{sec})$ | Col. 7 <br> Sum of <br> all pa- <br> (titions <br> $(\mathrm{sec})$ | Col. 8 <br> Minimi- <br> zed to- <br> gether <br> $(\mathrm{sec})$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| C432 | 2 | 345 | 568 | - | - | 913 | 827 |
| C499 | 2 | 526 | 285 | - | - | 811 | 701 |
| C880 | 4 | 32 | 55 | 69 | 90 | 246 | 218 |
| C1355 | 4 | 661 | 563 | 24 | 3375 | 4623 | 4537 |
| C1908 | 4 | 1013 | 386 | 34 | 34 | 1467 | 1178 |

Table 4.6. Efficiency of using inter-partition parallelism.

With inter-partition load balancing, there is a significant difference between the time needed to minimize the partitions simultaneously and consecutively. This shows that the inter-partition load balancing further enhances the overall efficiency achieved by intra-partition load balancing alone.

### 4.5. Comparison among MIS 2.1, SYLON-XTRANS, and PTRANS

In this section, comparisons are made among MIS 2.1 [BRSW87], SYLONXTRANS 1.1 [ X 90$]$, and PTRANS (our implemertation) on the Encore Multimax computer. PTRANS is executed on a single processor. The qualities of the final circuits are measured in terms of the number of simple gates and connections ( $\mathrm{g} / \mathrm{c}$ ). MIS 2.1 is executed on both partitioned and non-partitioned circuits using the Boolean script. The algebraic script is also used so as to demonstrate the effectiveness of don't-care based minimization in the Boolean script. In this script, the circuits are simplified using don't-cares and disjoint support filtering so that the final qualities can be compared with that of PTRANS as it is basically a don't-care bas $=$ d minimization program. In the following comparison tables, a $\because$ ' sign means either the corresponding program runs out of memory, could not finish within 30 hours, or unable to handle the number of inputs in the circuit.

Between XTRANS and PTRANS, the circuits Froduced by PTRANS are usually slightly bigger than those by XTRANS as shown in Table 4.7. This is because XTRANS 1.1 recognizes XOR and XNOR gates, which are presently not accepted by PTRANS. The timings of XTRANS is also faster than PTRANS by a factor of about 2

| Circuit | Initial$(\mathrm{g} / \mathrm{c})$ | Non-partitioned |  | Partitioned |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | MIS 2.1 <br> Algebraic ( $\mathrm{g} / \mathrm{c}$ ) | MIS 2.1 <br> Boolean $(\mathrm{g} / \mathrm{c})$ | No. of <br> partitions | MIS 2.1 <br> Boolean <br> ( $\mathrm{g} / \mathrm{c}$ ) | XTRANS <br> ( $\mathrm{g} / \mathrm{c}$ ) | PTRANS <br> ( $\mathrm{g} / \mathrm{C}$ ) |
| f51m | 131/270 | 107/233 | 110/225 | 1 | 110/225 | 70/128 | 81/157 |
| 5xpl | 129/279 | 103/224 | 92/185 | 1 | 92/185 | 62/112 | 79/153 |
| 9sym | 205/470 | 163/376 | 175/408 | 1 | 175/408 | 162/346 | 174/363 |
| bw | 205/481 | 142/312 | 123/250 | 1 | 123/250 | 144/262 | 144/288 |
| sao2 | 129/310 | 125/270 | 100/211 | 1 | 100/211 | 99/195 | 99/213 |
| vg2 | 158/391 | 71/147 | 66/141 | 1 | 66/141 | 82/158 | 73/162 |
| rd73 | 135/325 | 96/213 | 67/134 | 1 | 67/134 | 79/156 | 110/236 |
| duke2 | 485/1224 | 285/627 | 282/627 | 1 | 282/627 | 327/654 | 348/730 |
| alupla | 114/223 | 109/230 | 134/265 | 1 | 134/265 | 97/192 | 103/205 |
| misex 1 | 69/154 | 45/99 | 47/88 | 1 | 47/88 | 46/84 | 51/102 |
| misex 2 | 87/233 | 75/162 | 78/159 | 1 | 78/159 | 94/181 | 82/178 |
| misex3c | 493/1231 | 393/907 | 311/730 | 1 | 311/730 | 326/680 | 352/757 |
| C432 | 198/411 | - | - | 2 | 169/380 | - | 166/342 |
| C499 | 526/942 | 511/911 | 521/925 |  | 522/946 | - | 493/892 |
| C880 | 342/688 | 361/703 |  | 4 | 373/707 | - | 313/632 |
| C1355 | 492/1018 | 515/915 | 519/523 | 4 | 543/959 | - | 507/1022 |
| C1908 | 599/1220 | 528/983 | - | 4 | 554/1011 | - | 448/914 |

Table 4.7. Comparison of circuit qualities among MIS 2.1, XTRANS 1.1, and PTRANS.
to 3. However, as PTRANS uses the disk as a temporary storage, time is needed to pack and unpack the BDDs as they are transferred to and from the disk. This has been found at times to amount to greater than $50 \%$ of the total time taken by PTRANS. Hence, the actual time used by PTRANS in performing the Transduction procedures is much smaller than those shown in Table 4.8. Notz however that this feature was incorporated into PTRANS to handle very large circuits which cannot be handled by XTRANS 1.1.

An interesting point to note is the difference in time PTRANS and XTRANS take for the circuit alupla. PTRANS is about 20 times fas:er than XTRANS. This could be due to the difference between the BDD- and the SOP- representations used by the two.

| Circuit | Non-partitioned |  | Partitioned |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | MIS 2.1 <br> Algebraic <br> Time <br> (sec) | MIS 2.1 <br> Boolean <br> Time <br> (sec) | No. of partitions $(\mathrm{sec})$ | MIS 2.1 <br> Boslean <br> Tine <br> (sec) | XTRANS <br> Time <br> (sec) | PTRANS <br> (1 proce- <br> ssor) <br> Time <br> (sec) |
| f51m | 37 | 77 | 1 | 77 | 103 | 217 |
| 5xpl | 35 | 59 | 1 | 59 | 86 | 182 |
| 9 sym | 139 | 150 | 1 | 150 | 215 | 5408 |
| bw | 56 | 265 | 1 | 255 | 331 | 793 |
| sao2 | 41 | 40 | 1 | 40 | 371 | 1556 |
| vg2 | 28 | 165 | 1 | 165 | 626 | 1603 |
| rd73 | 50 | 33 | 1 | 33 | 562 | 967 |
| duke2 | 159 | 7897 | 1 | 7897 | 9801 | 19650 |
| alupla | 26 | 361 | 1 | 261 | 111420 | 6560 |
| misex1 | 10 | 9 | 1 | ç | 33 | 115 |
| misex2 | 15 | 65 | 1 | ¢j | 176 | 264 |
| misex3c | 379 | 7776 | 1 | 7776 | 24065 | 72834 |
| C432 | - | - | 2 | 1284 | - | 4911 |
| C499 | 86 | 9842 | 2 | 1;'69 | - | 5231 |
| C880 | 79 | - | 4 | 380 | - | 1535 |
| C1355 | 86 | 9343 | 4 | 559 | - | 33517 |
| C1908 | 2599 | - | 4 | 3451 | - | 6219 |

Table 4.8. Comparison of timings among MIS 2.1, XTRANS 1.1, and PTRANS on the Encore Multimax with a single processor.

A limitation of XTRANS 1.1 is that it only manages circuits with 32 inputs or less. Hence, it is unable to minimize any of the ISCAS circuits. However, XTRANS 2.0 is currently being developed and will avoid this limitation.

The circuits used to compare between XTRAN and PTRANS are also used for MIS 2.1 and PTRANS. The MCNC circuits can be minimized without partitioning. The qualities produced by both MIS 2.1 and PTRANS for these circuits are comparable and MIS 2.1 is faster than PTRANS.

However, MIS 2.1 is unable to handle larger circuits such as C432. Such circuits are partitioned and both programs are executed on the ${ }^{\eta}$ je partitions. As Table 4.7 shows, PTRANS produces better qualities than MIS 2.1 consistently. In fact, some of these
final circuits are even smaller than those produced $\mathrm{b} y$ MIS 2.1 running on the original circuits as a whole.

Although PTRANS is usually slower than MIS 2.1 , the advantage with using PTRANS is that it is parallelizable as can be seen from the results in the previous sections. On the other hand, [Z91] has already shown that MIS is very difficult to parallelize. Hence, when multiple processors are employed, PTRANS is able to both execute faster and produces better quality circuits than MIS 2.1.

## CHAPTER 5.

## CONCLUSIONS

In this thesis, a parallel algorithm implementing the Transduction Method has been proposed and implemented. In Chapter 1, an introduction to the problem of multi-level logic synthesis is given. Chapter 2 shows the methods of computing the MSPFs and CSPFs of simple gates. The basics of SYLON-XTRANS is also summarized. It contains four main transformation procedures, name $y$, pruning, gate substitution, gate merging and generalized gate substitution/gate input reduction. More detailed information on these procedures can be found in [X90].

The parallelization of XTRANS is described in Chapter 3. The major problem PTRANS faces is the need for large input circuits to achieve high processor utilization. This is limited by the amount of memory availiable on our computer system. To solve this, large circuits have to be partitioned into smaller components. PTRANS also needs to manage the disk as a temporary storage. Wher. multiple partitions are present, PTRANS is able to minimize them simultaneously, or consecutively. The consecutive minimization of partitions is also performed in parallel. These intra- and inter-partition parallelisms are achieved through the multiple mister-slave program model used. Furthermore, PTRANS also uses BDDs instead of the SOP representation. BDDs are generally more compact the the latter in representing Eoolean functions.

With both types of parallelism, the results prodiced by PTRANS are presented in Chapter 4. Considering the efficiency of the implementation of PTRANS, it has achieved good speedups and high processor utilization, even when not using interpartition dynamic load balancing. Of course, when nter-partition parallelism is used, the efficiency achieved is even higher. As compared with XTRANS, PTRANS has comparable performance. However, when executing on a single processor, it is slower than MIS 2.1 due to differences in algorithm complexities. On the other hand, PTRANS is parallelizable and produces better quality circuits than MIS 2.1.

On the whole, the PTRANS implementation hes been very successfully. Future work includes porting it to the Chare Kernel Progamming Language, which is also developed at University of Illinois. This language is machine-independent, and will allow PTRANS to execute on most of today's paralle machines with little or no source changes.

## REFERENCES

S. B. Akers, "Binary Decision Diagrams," IEEE TC, 1978, pp. 509-516.
R. K. Brayton, et al., "ESPRESSO-II: A New Logic Minimizer for Programmable Logic Arrays," CICC, Jane 1984, pp. 370-376.
R. Bryant, "Graph-Based Algorithms for Boolean Functions Manipulation," IEEE TC, Aug., 1986, pp. 677-691.
[DBK90] S. Dey, F. Berglez, and G. Kedem, "Dorolla Based Circuit Partitioning and Resynthesis," 27th DAC, 1990, pp: 607-612.
[BRSW87] R. K. Brayton, R. Rudell, A. S. Vincentelli, and A. R. Wang, "MIS: A Multiple-level Logic Optimization System," ICCAD, Nov., 1987, pp. 1062-1081.
[C87] K. C. Chen, "Program PMIN for PLA Minimization," M.S. thesis, Dept. of Computer Science, Univ. of Ill., Urbana, 1987.
[CHNS88] H. Cho, G. Hachtel, M. Nash, and L. Setiono, "BEAT-NP: A Tool for Partitioning Boolean Networks," ICCA"), 1988, pp. 10-13.
[CM89] K.C. Chen, and S. Muroga, "SYLON-DREAM : A Multi-level Network Synthesizer," ICCAD, 1989, pp. 552-5.55.
[DB91] K. De, and P. Banerjee, "Logic Partitioning and Resynthesis for Testability," ITC, 1991.
[FFK88] M. Fujita, H. Fujisawa, and N. Kawat:, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams,"

ICCAD, Nov., 1988, pp. 2-5.
R. Galivanche, "A Parallel Logic Minimization Algorithm for PLA Synthesis," M.S. thesis," Univ. of Iowi , 1986.
M. R. Garey, and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," San Fransico, CA, W. H. Freeman \& Co., 1979.
[HMJ88]
[KL70] B. W. Kemighan, and S. Lin, "An Zfficient Heuristic Procedure for Partitioning Graphs," Bell System Technical Journal, vol. 49, 1970, pp. 291-307.
[LM90] J. C. Limqueco, and S. Muroga, "SYLION-REDUCE : A MOS Network Optimization Algorithm using Permissible Functions," ICCD, 1990.
[MF89] Y. Matsunaga, and M. Fujita, "Multi-level Optimization using Binary Decision Diagrams," ICCAD, 1989, pp 556-559.
[MK89] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The Transduction Method - Design of Logiy Networks Based on Permissible Functions," IEEE TC, Oct., 1989, pp. 1404-1424.

Verification using Binary Decision Diagrams in a Logic Synthesis Environment," ICCAD, Nov., 1988, pp. 6-9.
[PBP89]
[PBP91]
[SB88]
[SB90]
[XM89] X. Q. Xiang, and S. Muroga, "SYLON-XTRANS:A Multilevel Logic Network Synthesizer," IWLS, NCMC, May 1989.
[X90] X. Q. Xiang, "Multi-Level Logic Network Synthesis System, SYLONXTRANS and Read-Only Memory Minimization Procedure, MINROM," Ph. D. thesis, Dept. of Computer Science, Univ. of Ill., Urbana, 1990.
[Z91] G. Zipfel, "Parallel Algorithm for Algebraic Factorization with Application to Multi-Level Logic Syntnesis," M.S. thesis, Univ. of Ill., Urbana, 1991.


