INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Fault tolerance and reconfiguration strategies for tree architectures

Ko, Chen-Ken, M.S.
The University of Arizona, 1990
FAULT TOLERANCE AND RECONFIGURATION STRATEGIES FOR TREE ARCHITECTURES

by

Chen-Ken Ko

A Thesis Submitted to the Faculty of the Department of Electrical and Computer Engineering
In Partial Fulfillment of the Requirements For the Degree of Master of Science With a Major in Electrical Engineering In the Graduate College The University of Arizona

1990
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:  

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Dr. Sy-Yen Kuo  
Assistant Professor of Electrical and Computer Engineering  

March 9, 1990

Date
ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Sy-Yen Kuo, my advisor and chairman of the Committee, for his guidance and support throughout this work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>6</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>7</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>8</td>
</tr>
<tr>
<td>2 PERFORMANCE DEGRADATION APPROACH</td>
<td>12</td>
</tr>
<tr>
<td>2.1 The algorithm by Horowitz and Zorat</td>
<td>12</td>
</tr>
<tr>
<td>2.2 The proposed algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Examples</td>
<td>17</td>
</tr>
<tr>
<td>3 BLOCK-ORIENTED BINARY TREE ARCHITECTURES</td>
<td>20</td>
</tr>
<tr>
<td>3.1 $G_1$ architecture</td>
<td>22</td>
</tr>
<tr>
<td>3.2 $G_2$ architecture</td>
<td>24</td>
</tr>
<tr>
<td>3.3 $G_3$ architecture</td>
<td>27</td>
</tr>
<tr>
<td>4 RECONFIGURATION ALGORITHMS</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Reconfiguration of $G_1$ architectures</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Reconfiguration of $G_2$ architectures</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Reconfiguration of $G_3$ architectures</td>
<td>33</td>
</tr>
<tr>
<td>5 EXTENSIONS TO $d$-ARY TREES</td>
<td>36</td>
</tr>
<tr>
<td>5.1 $G_1(d, L, k_l, b_l)$</td>
<td>36</td>
</tr>
<tr>
<td>5.2 $G_2(d, L, k_{sib}, b_s)$</td>
<td>39</td>
</tr>
</tbody>
</table>
5.3 \( G_3(d, L, k_{\text{zib}}, b_i) \) ................................................. 40

5.4 Reconfiguration of \( d \)-ary trees ........................................... 41

6 EVALUATION OF RELIABILITY AND YIELD .......................... 44

7 DISCUSSION AND CONCLUSION ........................................ 53

APPENDIX A ................................................................. 57

REFERENCES ............................................................... 71
LIST OF FIGURES

2.1 A binary tree in the paper of Horowitz and Zorat. .................. 13
2.2 A full ring binary tree. ........................................... 17
3.1 Switch structures. .................................................. 21
3.2 $G_1(3, k_1, b_1)$. .................................................. 24
3.3 An H-tree layout outline of a four level $G_1$. ....................... 25
3.4 A three level implementation of $G_2$. ............................ 26
3.5 An H-tree layout outline of a four level $G_2$. ....................... 27
3.6 Example implementations of $G_3$. ................................ 30
4.1 Reconfiguration of $G_1$ with faulty PEs (crossed). ................ 33
4.2 Reconfiguration of $G_2$ with faulty PEs (crossed). ................ 34
4.3 Reconfiguration of $G_3$ with faulty PEs (crossed). ................ 35
5.1 Outlines of three level 3-ary trees. ............................... 37
6.1 Reliability comparison of eight level binary trees. ................. 48
6.2 NRIF comparison of eight level binary trees. ....................... 49
6.3 Yield comparison of eight level binary trees. ...................... 51
6.4 CAUF comparison of eight level binary trees. ...................... 52
ABSTRACT

Reconfigurable binary tree architectures have been widely studied and used in various VLSI implementations. These fault tolerance approaches can be classified into two categories. In this thesis, we propose a fault diagnosis for the first category. Then a new block-oriented fault tolerance scheme for tree architectures is presented for the second category. The fundamental idea is to extend each single PE node in the tree to a block. Each block could consist of several PEs and the associated interconnection links. It is shown that several previous fault tolerant designs in the literature are special cases of the proposed design. The VLSI layout of binary tree is very efficient and the problem of long interconnections in other designs has been alleviated. Efficient reconfiguration algorithms and reliability analysis are also presented.
CHAPTER 1

INTRODUCTION

The binary tree architecture is a very useful interconnection structure for parallel computing systems [1-5]. It reduces the intercommunication complexity of a $n$-node system from $O(n)$ to $O(\log n)$ which is very significant in VLSI systems with a large number of PEs. It also has the well known H-tree and other efficient VLSI layout strategies [6,7]. The weakness, however, is that one single failure in a node or a link in the tree could malfunction the whole system. Therefore, many fault tolerance schemes were introduced to support a reliable and applicable system. These previous approaches can be classified into two categories. Approaches in the first category [3] allow performance degradation by including redundant links in the system to skip faulty nodes and to reroute communication paths in case of faults. Approaches in the second category [8-16] retain the basic tree structure of original size by adding not only redundant links but also extra nodes for fault tolerance reconfiguration in the presence of faults.

The first category was addressed by Horowitz and Zorat [3]. In Chapter 2 we also study this problem. The basic fault tolerant design is to construct a half-ring,
full-ring, hypertree [5] or other tree architectures using extra links. Finding the communication path with the least cost between any pair of nodes before and after appearance of faults is the major concern for these strategies. In our development, the condition flags are applied to construct the tree structure and to indicate the current status of nodes and links. The communication path between every pair of nodes is stored in a global table. An algorithm is implemented to search the shortest communication paths between nodes before and after failures of nodes and links. However, these approaches destroy the original tree structure which is not desirable to many applications requiring the entire performance of the whole rigid tree [4].

For the second category, Hayes [9] first proposed an optimal 1-fault tolerant tree model which could not tolerate multiple faults. Kwan and Toida [10] derived an optimal 2-fault tolerant design for the symmetric tree structure. Raghavendra, Avizienis and Ercegovac [11] presented a scheme which can tolerate multiple faults, but with problems such as large number of long redundant links and less efficient VLSI layout. The elegant idea of a modular approach was first presented by Hassan and Agarwal [12] and improved by Singh [13]. Its VLSI layout may be more efficient than other previous contributions, but was specific for a particular case and could not suggest a simple VLSI layout to the extension schemes. In addition, the PEs of a module in Singh’s design actually occupy two levels of a tree instead of one. The reconfiguration needs to be started from the lowest level to the root to obtain maximum reliability and yield enhancement. The SOFT scheme by Lowrie and Fuchs [14] tried to take advantage of the existing VLSI layout by having less link redundancy but with a more complicated reconfiguration process, especially when
a fault occurs at a level near the root. The cluster model by Hosseini, Kuhl and Reddy [15] is even more complicated to be implemented.

Recently, Dutt and Hayes [16] presented a scheme to optimally generalize the k-F-T problem at each level with k spares per level and the smallest number of redundant links. However, the link redundancy of their design is not enough. For example, the 2-FT NST \( G_4[2, T_N(3, 3)] \) in Fig. 6 of their paper [16] will not succeed in reconfiguration if \( x_{0,0}, x_{0,1}, x_{1,2} \) and \( x_{1,3} \) fail. We also do not think it is appropriate to have the same fault tolerance capability of k in every level for the following two reasons. First, if k is too large, there exist so many unused redundant nodes and links in the upper levels of the tree, especially in the levels near the root. Second, if k is too small, the redundancy is not enough at the lower levels where many nodes can be faulty as the total number of nodes in these levels becomes larger and larger. In addition, their scheme needs a very complex covering graph as well as sophisticated reconfiguration algorithms, and still they did not propose a good method to implement their structures and solve the long link problem. They did, however, reveal a good terminology to the study of the fault tolerant tree architectures.

In Chapters 3 and 4 of this thesis, we present a novel block-oriented architecture, its implementation and reconfiguration strategies for the fault tolerant tree structures in the second category. This model can reduce the long link problem and can be implemented efficiently using the existing embedding strategies of VLSI binary tree layout [6,7]. Chapter 5 extends this philosophy to general trees with variant numbers of PEs in a tree node. Chapter 6 evaluates the proposed scheme
and compare the result with that of Singh's scheme for binary trees. Discussions and comparisons with other designs are in Chapter 7.
CHAPTER 2

PERFORMANCE DEGRADATION APPROACH

The study in this category has the following assumptions: 1) the factor affecting this approach is restricted to be the number of links, other factors, such as lengths of links (result into the propagation delay) and congesting problems, can be added as a cost to links for the shortest path computing; 2) the probability of link and switch faults is negligible compared with that of PE faults as assumed in all other fault tolerant designs; this assumption is reasonable for sophisticated PE systems and for modeling link failures into the respective PEs or as coverage failures [13].

2.1 The algorithm by Horowitz and Zorat

In this approach [3], the tree nodes are numbered as in Figure 2.1. Only the leaves of the tree are connected with redundant links to reroute paths in case of faults. The algorithm is referred to as a graph traversal in depth-first manner. Because much of the structure of the graph is known in advance, the algorithm is designed to effectively exploit this knowledge. It includes features that test for subtree inclu-
The algorithm for processing a message is described as follows:

Input: a message, its original source node $i$, its destination node $j$, the current node $k$, the last previously visited node $\text{prev}(k)$, a list of quadruples of the form $(h, p, l, r)$, where $h$ is a node number and $p, l, r$ are Boolean values: $p(l$ or $r)$ is true if the message has already visited node $h$ and has being sent to the parent, (left child or right child) of $h$.

Output: either 1) "success" - destination reached; or 2) "failure" - there is no path from $i$ to $j$; or 3) the address of a neighbor of $k$ to which the message should be forwarded.

Procedure:

if $i = j$ then success, stop
if the node number in the last entered quadruple is not equal to \( k \)
then append \((k, false, false, false)\) to the list of quadruples

if \( k = i \) and \( p = l = r = true \) then failure, stop

else let \( x \) be this last entered quadruple
endif

if any of the neighbors other than prev(\( k \)) have already been visited
then mark its corresponding component of \( x \) as true
endif

if any of the links \( p, l, r \) of \( k \) is faulty
then mark its corresponding component of \( x \) as true
endif

choose a direction using the following strategy

\( \text{prev}(k) \) is chosen last

if \( j \) is in a subtree of \( k \) and the corresponding link of \( k \) is not faulty
then choose that link
else

if \( j \) is left of \( k \) and left(\( k \)) is not faulty
then choose the link labeled \( l \)
endif

if \( j \) is right of \( k \) and right(\( k \)) is not faulty
then choose the link labeled \( r \)
endif

mark the component of \( x \) corresponding to the chosen link as true

send the message over the chosen link
If paths between the source and destination exist, this algorithm will find one of them; otherwise, it reports *failure*. However, they did not discuss whether it is one of the shortest communication paths. Meanwhile, in the worst case, a node is visited three times by any given message and each time the node will execute the algorithm above. This requires the inspection of the list of the quadruples, which takes time proportional to \( n \), the number of nodes in the tree. The entire routing of the message from the source to the destination corresponds to a graph traversal for which the worst case complexity is linear in the number of edges. For the particular case of the completely linked full binary tree, the number of edges is \([3n/2] - 1\), giving an overall complexity for the routing of a message from source to destination of \( O([3n/2]) \cdot O(n) = O(n^2) \).

2.2 The proposed algorithm

In our development, we renumber a complete full ring binary tree as in Figure 2.2. Each tree node is represented by \((p, bl, br, cl, cr, pre, length, visit)\), an eight element structure, where \(p, bl, br, cl\) and \(cr\) are Boolean flags to represent the status of path to its parent, left brother, right brother, left child, and right child respectively. The combination of these flags characterizes a full ring, a half ring, or a hyper tree architecture. \(pre, length\) and \(visit\) are used to determine the shortest path between two nodes. Every shortest communication path between any two nodes are stored in a matrix table. A row number of the matrix table represents a source node;
while a column number is a destination node. Every member is a linked list whose nodes store node numbers traveled on this path. The basic idea of constructing the table is specified as follows: 1) using the shortest path algorithm to compute the matrix table of communication paths for the tree structure; 2) updating the condition flags and reporting failures of members of the routing table which are directly influenced by faulty nodes and links; 3) reporting the knowledge if the path between the source and destination is not influenced or failed; 4) otherwise, computing a new one using the shortest path algorithm; if a shortest path is found, it replaces the corresponding member in the matrix table; otherwise, the member is reported as failed.

The fundamental concept of determining the shortest path is similar to the one by Dijkstra [8]. It calls a procedure to compute the possible shortest paths starting from the source node \(i\) to a node \(k\) which is connected but good. If none is found, it stops the searching and reports failure, else it selects the next visited node. This procedure will continue until a shortest path connected to the destination node \(j\) is found or a failure is reported. If a path is found, it calls procedures (create and reverse) to produce routes between \((i, j)\) and \((j, i)\) respectively. This algorithm guarantees to find one shortest path from existing ones. In the worst case, the complexity is \(O(n) \cdot O(n - 2)\), which is still \(O(n^2)\). A detailed implementation of the entire algorithm for a complete full ring binary tree using this performance degradation strategy is in Appendix.
2.3 Examples

These two algorithms seem to base on tree structures with different amount of link redundancy. However, as mentioned above, the algorithm presented in section 2.2 uses flags to represent every link to a node, it should be used by any kind of tree structures, even for a $d$-ary tree architecture. But the corresponding flags of the implementations should be able to represent every link of a tree structure. In this section, we compare the results of two examples in the paper by Horowitz and Zorat for both algorithms. Additional examples are also transcribed for our algorithm. These examples are based on a 5-level binary tree.

Example 1 by Horowitz and Zorat:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>3</td>
<td>1, 2, 5, 6</td>
</tr>
</tbody>
</table>
Example 1 by the proposed algorithm:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>1, 2, 5, 6</td>
</tr>
</tbody>
</table>


Example 2 by Horowitz and Zorat:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>3</td>
<td>1, 5, 7, 16, 17</td>
</tr>
</tbody>
</table>

(21, 1, 0, 0) (13, 1, 0, 1) (29, 0, 1, 1) (19, 1, 0, 0) (15, 1, 0, 1) (27, 1, 1, 1) (13, 1, 1, 1) (25, 1, 1, 1) (21, 1, 1, 1) Failure.
Example 2 by the proposed algorithm:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>3</td>
<td>1, 6, 7, 16, 24</td>
</tr>
</tbody>
</table>


Example 3 by the proposed algorithm:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>28</td>
<td>12, 19, 22, 24</td>
</tr>
</tbody>
</table>

4 — 2 — 3 — 7 — 14 — 28 — Success.

Example 4 by the proposed algorithm:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>15</td>
<td>4, 9, 19</td>
</tr>
</tbody>
</table>

Failure.
CHAPTER 3

BLOCK-ORIENTED BINARY TREE ARCHITECTURES

The block-oriented reconfigurable binary tree architecture extends each processing element (PE) in a nonredundant tree to a block. In addition to the original PE, each block could consist of several spare PEs and associated interconnection links. Faulty PEs and links can then be replaced by appropriate spares. This idea evolves into three basic designs, $G_1$, $G_2$ and $G_3$, based on their fault tolerance capabilities, link constructions and switch structures. The detail implementations of PE allocation, link connection and switch utilization are influenced by the number of spare PEs in each block. However, fundamental frameworks are similar.

In the following, a PE in a fault-tolerant tree can be either a tree PE or a spare PE. A PE of a block corresponding to a node in a nonredundant original tree architecture is called a tree PE, otherwise it is a spare PE. Links are specified into three categories: 1) down-link, used only by a PE to communicate with its children; 2) up-link, used by a PE to communicate with its parent; 3) intermediate-link, which can be either an up-link or a down-link at different times. A switch structure is
represented as $S_{u,d,i}$, where $u$, $d$ and $i$ are the numbers of up-links, down-links and intermediate-links connected to the switch $S$ respectively. For example, Figure 3.1 shows the switch structures used in the proposed $G_1$ and $G_2$ architectures. In these designs, only a pair of links in a switch can be used at the same time to construct a path. The possible combinations are an up-link and a down-link, an up-link and an intermediate-link as a down-link, and a down-link and an intermediate-link as an up-link.
3.1 $G_1$ architecture

The first $G_1$ design aims at optimal utilization of spare PEs at the same level of a binary tree. The levels of a L-level tree are numbered from 1 to $L$ starting at the root.

**Definition 1** Let $G_1(L, k_l, b_l)$ denote a $L$-level binary tree, where each node $i$ of level $l$ is a block $B_i$ consisting of $b_l$ PEs and associated links. Each level $l$ has $k_l$ spare PEs such that it can tolerate up to $k_l$ faulty PEs and preserve the original tree structure.

The number of PEs at level $l$ in a nonredundant complete binary tree is denoted as $n_l = 2^l - 1$. Therefore, the total number of PEs in this tree from level 1 to level $L$ is $N_l = \sum_{i=1}^{l} n_i = \sum_{i=0}^{l-1} 2^i$. A node $i$ is numbered in breadth-first manner from left to right in a level and from level 1 to level $L$. (In the following discussion, all numberings are assumed in breadth-first order from left to right and from root to leaf.) $k_l$ is the number of spare PEs (fault tolerance capability) of level $l$. Block $B_i$ groups $b_i$ PEs together as the $i$-th node in the redundant tree architecture.

**Lemma 1** In order to let $G_1(L, k_l, b_l)$ have a fault tolerance capability of $k_l$ for each level from $l = 1$ to $L$, in addition to the original paths in a nonredundant tree, paths should exist between a tree PE in $B_i$ at level $l$ ($\leq L - 1$) and a spare PE in all blocks at level $l + 1$ as well as between a spare PE in $B_i$ and a PE in all blocks at level $l + 1$.

**proof:** For $k_l$ spare PEs at each level $l$ to achieve up to $k_l$ fault tolerance
capacity, every spare should be able to replace any faulty PE at level $l$. This means there should exist paths between every spare PE of level $l$ and its parent as well as the children of any PE at the same level. Adding redundant paths between a tree PE in $B_i$ at level $l$ ($\leq L - 1$) and a spare PE in all blocks at level $l + 1$, paths between every spare PE of level $l$ and its parent are achieved. While with extra paths between a spare PE in $B_i$ and a PE in all blocks at level $l + 1$, paths between every spare PE and the children are set up. Therefore, the construction of paths as in Lemma 1 is needed to have a fault tolerance capability of $k_l$ for each level from $l = 1$ to $L$.

In our $G_1$ design, we will restrict the number of PEs in blocks as $b_1 = 2, b_{2j} = 2$ and $b_{2j+1} = 1$ for $j \geq 1$. This results in $k_{l=1} = 1$ and $k_l = n_{l-1}$ for $l \geq 2$. The architectures with variant $b_i$ values will be discussed in latter sections. Figure 3.2 shows an example design, $G_1(3, k_l, b_i)$. Links and switch structures are designed to catch the requirement of paths in Lemma 1 as well as to reduce long links if possible. $T_{i,1}$ is the regular PE and $T_{i,2}$ is a spare PE in block $B_i$. The heavy lines are the current active communication paths and light ones are spare interconnection links. (It is not necessary to have spare links in a block at the lowest level.) The switches utilized are specified by heavy dots. Note that only three types of blocks, $B_1, B_{2j}, B_{2j+1}(j \geq 1)$, are needed for $G_1$ in Figure 3.2. Hence, once we have designed these three blocks, they can be used repeatedly to construct a large tree. This property is very desirable in VLSI implementation. An example H-tree layout for four levels $G_1$ is outlined in Figure 3.3, which keeps a sibling set as a unit because each sibling set has a similar structure. (A sibling set represents a set of blocks with the same parent block)
However, as can be seen in Figure 3.3, there exist long links in lower levels which are not desirable in VLSI implementation, especially for large trees. Another design defined below is introduced to alleviate this problem. The objective of this $G_2$ architecture is to let spare PEs of a block tolerate faults within the sibling set of this block only.

**Definition 2** Let $G_2(L, k_{sib}, b_i)$ denote a $L$-level binary tree, where each node $i$ is a block $B_i$ consisting of $b_i$ PEs and associated links. Each sibling set $sib$ has $k_{sib}$ spare PEs such that it can tolerate up to $k_{sib}$ faulty PEs without destroying the
Lemma 2 In order to let each sibling set have a fault tolerant capacity up to $k_{\text{sib}}$ in $G_2$, in addition to the original paths in a nonredundant tree, appropriate paths should exist between a tree PE in block $B_i$ and a spare PE in all child blocks of $B_i$ as well as between a spare PE in $B_i$ and a PE in child blocks of $B_i$ and of the other sibling block of $B_i$.

The terminology of this alternate design is similar to those of $G_1(L, k_l, b_l)$. The major changes are: 1) $\text{sib}$ is used to number sibling sets from root to leaf; 2) the number of spare PEs and fault tolerance capability is now $k_{\text{sib}}$ for sibling set $\text{sib}$ instead of $k_l$ for level $l$. Therefore, the number of sibling sets at level $l$ in a binary tree is denoted as $n_{l-1}^{\text{sib}} = n_{l-1}$, where $n_1^{\text{sib}} = 0$ for the root. The total number of sibling sets in a binary tree from level 1 to level $l$ is denoted as $N_l^{\text{sib}} = \sum_{i=1}^{l} n_i^{\text{sib}} = N_{l-1}$,
where $N_{sib} = 0$. The proof of Lemma 2 which is not shown is similar to that of Lemma 1. In the $G_2$ design, we restrict the number of PEs in a block to be $b_i = 2$. This results in $k_{sib} = 1$ and $k_{sib} = 2$, $\forall sib \geq 1$. An example implementation is in Figure 3.4, whose links and switch structures are designed to fulfill the requirement of Lemma 2. (It is not necessary to have spare links in a block at the lowest level.) Figure 3.5 is the outline of the H-tree layout for a four level $G_2$. Comparing Figure 3.3 and Figure 3.5, we can see that long links have been eliminated in $G_2$. 

Figure 3.4: A three level implementation of $G_2$. 

Figure 3.5: A four level $G_2$. 

--- 

26
Figure 3.5: An II-tree layout outline of a four level $G_2$.

3.3 $G_3$ architecture

If the number of spare PEs in a sibling set of $G_2(l, k_{sib}, b_i)$ is small, the reliability and yield enhancement may not be acceptable. But if the number of spare PEs is large, the cost may be too high. In this section, we further modify $G_2$ into $G_3$ to improve this situation by adding minimum redundant paths and making a slight change of the reconfiguration algorithm.

Definition 3 Let $G_3(L, k_{sib}, b_i)$ denote a $L$-level binary tree, where each node $i$ is a block $B_i$ consisting of $b_i$ PEs and associated links. Each sibling set $sib$ has $k_{sib}$ spare PEs. In addition to the maximum fault tolerance capability $k_{sib}$ of each sibling set as in $G_2$, faults can also be tolerated across neighboring levels without destroying the original tree structure.
Lemma 3  In addition to the fault tolerance capability as in Lemma 2, in order to let a spare PE in each sibling set tolerate a faulty PE in its parent block, the following paths should be added: 1) between a spare PE (serving as a parent if needed) and other PEs within a sibling set; 2) a bypass structure in each PE to construct paths between a PE in $B_i$ at level $l$ and a spare PE in grandchild blocks of $B_i$ at level $l+2$ for $l \leq L - 2$ (serving as left or right child, in case of skipping a block at level $l+1$).

Lemma 4  In addition to the fault tolerance capability as in Lemma 2 and Lemma 3, in order to let a spare PE in each sibling set tolerate a faulty PE in its child blocks as well as one in its parent block, the following paths should be added: 1) between a tree PE and a spare PE (serving as a child if needed) within a block; 2) between a spare PE in block $B_i$ at level $l$ and a PE in grandchild blocks of $B_i$ at level $l+2$ for $l \leq L - 2$ (in order to skip a block at level $l+1$).

In $G_3$, the definitions of terms, such as $b_i, n_i^{\text{sib}}, N_l^{\text{sib}}, n_s, n_l$ and $N_l$ are the same as those in $G_2(L, k_{\text{sib}}, b_i)$. The proofs of Lemma 3 and Lemma 4 are also similar to that of Lemma 1. But the fault tolerance capability of each sibling set with $k_{\text{sib}}$ spares is different since the additional links add the recovery ability across adjacent levels. Besides, because a faulty PE at level $l$ might be replaced by a spare PE at level $l + 1$ or $l - 1$, a PE in $G_3$ includes a bypass structure. In the $G_3$ design, the same PE placement as in $G_1$ is used, i.e., $b_i$ are the same. Figures 3.6(a) and (b) show the implementations to catch the requirements of Lemma 3 and Lemma 4 respectively. Their difference is that some extra links are needed in Figure 3.6(b). However, these links let a spare PE of block $B_i$ at level $l$ tolerate a faulty PE in
either the parent block of $B_i$ at level $l-1$ or child blocks of $B_i$ at level $l+1$, instead of a faulty PE at level $l-1$ only as in Figure 3.6(a). The outlines of H-tree layout for Figures 3.6(a) and (b) are the same as in Figure 3.5.
Figure 3.6: Example implementations of $G_3$. 
CHAPTER 4

RECONFIGURATION ALGORITHMS

In this section, we present reconfiguration algorithms for the $G_1$, $G_2$ and $G_3$ architectures. All these three algorithms use the following PE, block and system condition flags to control the recovery process:

- PE conditions: *good* and *fail*
- block conditions: *hasspare*, *nospare* and *fail*.
- system conditions: *good* and *fail*.

4.1 Reconfiguration of $G_1$ architectures

In the beginning, conditions of all PEs and the system are assumed to be *good*. The condition of a block is *hasspare* or *nospare* depending on whether there is a spare PE in this block or not. Then it is necessary to investigate the status of each tree PE after fault diagnosis. The investigations follow the breadth-first order from root to leaf. If a fault appears at a tree PE, all paths connected to it are cut off. If the current status of the block in which this faulty PE resides is *nospare*, it is
changed to *fail*. Otherwise, the spare PE is investigated. If the spare is *good*, it replaces the faulty one. The status of the block becomes *nospare*. If it is *fail*, all paths connected to it are cut off and the status of the block becomes *fail*. After the process of recovery within each block for all faulty PEs, a recovery by other spares at the same level is performed from level 1 to level $L$ if there is still any *fail* block. In this procedure, a loop is used to find a *has spare* block at the same level for every *fail* block. If one is found, it does a recovery process similar to the recovery within a block. A *good* spare will replace the faulty one. Both blocks become *nospare*. If no *good* spare exists, the system is reported *fail* and the reconfiguration terminates. An example reconfiguration of $G_1$ is shown in Figure 4.1 where heavy lines are new communication paths after excluding the faulty PEs (crossed) and cutting off their original communication paths.

### 4.2 Reconfiguration of $G_2$ architectures

The reconfiguration of $G_2$ has a similar process as that of $G_1$, but the number of spare PEs, $k_{sib}$, is now restricted to each sibling set $sib$. That is, only the spare PEs within the sibling set are used to perform the further recovery, after the process of recovery within each block for all faulty PEs. As mentioned in section 2.1, the total number of sibling sets for a $L$ level tree is $N_L^{sib} = N_{L-1}$, if we let the root be the sibling set $0$. Therefore, the proceeding of this recovery out of a block is called for each sibling set from $sib = 1$ to $N_L^{sib}$ if a block is *fail*. Figure 4.2 is an achievement instance of the reconfiguration algorithm for $G_2$. 
4.3 Reconfiguration of $G_3$ architectures

In this section, we present reconfiguration algorithms for both implementations of the $G_3$ architecture. Each strategy is designed to have a better utilization of spare PEs. For the implementation in Figure 3.6(a), the algorithm consists of the following steps for each level from 1 to $L$ in breadth-first manner: 1) performing the recovery in a block; 2) using spare PEs in other sibling blocks and performing the recovery out of a block; 3) using spare PEs in child blocks of the failed block to proceed with the recovery out of a block (except the leaf blocks since they do not have child blocks). For the implementation in Figure 3.6(b), the strategy is illustrated as following for each level from 1 to $L$ in breadth-first manner: 1) using spare PEs.
in the parent block to proceed with the recovery out of a block (except the root block since it has no parent); 2) performing the recovery in a block; 3) using spare PEs in other sibling blocks and performing the recovery out of a block; 4) using spare PEs in child blocks of the failed block to proceed with the recovery out of a block (except the leaf blocks since they do not have child blocks). The example in Figure 4.3 applies the algorithm to Figure 3.6(a) which will be compared with Singh’s scheme [13] in Chapter 6.
Figure 4.3: Reconfiguration of $G_3$ with faulty PEs (crossed).
CHAPTER 5

EXTENSIONS TO \(d\)-ARY TREES

The three block-oriented fault tolerance approaches can be applied to general \(d\)-ary trees with variant \(b_i\) values. We specify them as \(G_1(d, L, k_i, b_i)\), \(G_2(d, L, k_{sib}, b_i)\) and \(G_3(d, L, k_{sib}, b_i)\). Terminology, construction and reconfiguration are fundamentally the same as in Chapter 2. Figure 5.1 shows outlines of a 3-ary tree for each of the \(G_1, G_2\) and \(G_3\) designs.

5.1 \(G_1(d, L, k_i, b_i)\)

In general, \(k_i\) and \(b_i\) are non-negative integers for all \(l\) and \(i\) such that \(k_i\) is equal to the sum of \((b_i - 1)\) for all \(i\) at the same level \(l\). Ideally, every block \(B_i\) should have nearly the same number of PEs since this will make the layout more efficient. For \(G_1\) architectures, the desirable \(b_i\) in level \(l\) will be only one value, \(b^l\), or at most two values, \(b^l\) and \(b^l - 1\). The following procedures are used to generate nodes and paths for \(G_1\). First, TREE-NODE-G1 decides the number of spare PEs in a sibling set based on the number of spare PEs at each level. Then it calls PLACE-PE to place...
Figure 5.1: Outlines of three level 3-ary trees.
PEs into all blocks. TREE-PATII-G1 corresponds to the requirement of paths in Lemma 1.

procedure TREE-NODE-G1();
begin
  \( k_{sib} = 0 := k_{l=1} \);
  for \( l := 2 \) to \( L \) do
    if \( k_l \mod n_l^{sib} = 0 \) then begin
      \( k := k_l/n_l^{sib} + 1; \)
      for \( sib := n_l^{sib} + 1 \) to \( n_l^{sib} \) do \( k_{sib} := k; \)
    end;
  else begin
    \( k := \lceil k_l/n_l^{sib} \rceil + 1; \)
    \( s := k_l \mod n_l^{sib}; \)
    for \( sib := n_l^{sib} + 1 \) to \( n_l^{sib} + s \) do \( k_{sib} := k; \)
    for \( sib := n_l^{sib} + s + 1 \) to \( n_l^{sib} \) do \( k_{sib} := k - 1; \)
  end;
PLACE-PE();
end; \{ TREE-NODE-G1 \}

procedure PLACE-PE();
begin
  for \( i := 1 \) to \( N_L \) do place \( T_{i,1} \) into \( B_i \);
  for \( sib := 1 \) to \( N_L^{sib} \) do
    if \( k_{sib} \mod d = 0 \) then
      for \( i := n_{sib-1} + 1 \) to \( n_{sib-1} + d \) do \( b_i := b_{sib} := k_{sib}/d + 1; \)
else begin

\( b^{sib} := k_{sib}/d + 1; s := k_{sib} \mod d; \)

for \( i := n_{sib-1} + 1 \) to \( n_{sib-1} + s \) do \( b_i := b^{sib}; \)

for \( i := n_{sib-1} + s + 1 \) to \( n_{sib-1} + d \) do \( b_i := b^{sib} - 1; \)

end;

for \( i := 1 \) to \( N_L \) do

if \( b_i \geq 2 \) then place \( T_{i,2}, T_{i,3}, \ldots, T_{i,b_i} \) into \( B_i; \)

end; \{ \text{PLACE-PE} \}

\[
\text{procedure TREE-PATH-G1();}
\]

begin

for \( l := 1 \) to \( L - 1 \) do

for \( i := N_{l-1} + 1 \) to \( N_l \) do begin

connect \( T_{i,x} \) and every PE in a child block of \( B_i; \)

connect \( T_{i,x} \) and other spare PEs at level \( l + 1; \)

connect \( T_{i,y} \) and other tree PEs at level \( l + 1; \)

\{ \( 1 \leq x \leq b_i \), \( 2 \leq y \leq b_i \} \}

end;

end; \{ \text{TREE-PATH-G1} \}

5.2 \( G_2(d, L, k_{sib}, b_i) \)

As discussed for \( G_1(d, L, k_i, b_i) \), the number of PEs \( b_i \) in a block \( B_i \) can only be one or two values for an efficient VLSI layout. In addition, if we look into the \( G_2 \)
architecture more carefully, it can be found that it is just like to build a spare tree along with the nonredundant one if every $b_i$ equals to 2, and to duplicate two spare trees if every $b_i$ equals to 3, and to build three spare trees if every $b_i$ equals to 4, • • •, and so forth. In addition to PLACE-PE in TREE-PATH-G1, TREE-PATH-G2 below is used to construct $G_2$ architectures. ($k_{sib}$ is given)

procedure TREE-PATH-G2();
begin
  for $l := 1$ to $L - 1$ do
    for $i := N_{i-1} + 1$ to $N_i$ do begin
      connect $T_{i,x}$ and every PE in a child block of $B_i$;
      connect $T_{i,x}$ and other spare PEs in child blocks of a sibling block of $B_i$;
      connect $T_{i,y}$ and other tree PEs in child blocks of a sibling block of $B_i$;
      \{ $1 \leq x \leq b_i$ \}, \{ $2 \leq y \leq b_i$ \}
    end;
end; \{ TREE-PATH-G2 \}

5.3 $G_3(d, L, k_{sib}, b_i)$

For general $G_3$ architectures, we still use the procedure PLACE-PE to construct the nodes but use new procedures TREE-PATH-G3-A and TREE-PATH-G3-B shown below to construct paths. It first uses TREE-PATH-G2 to place the paths as in $G_2(L, k_{sib}, b_i)$ structure, then it adds extra paths as in Lemma 3 and Lemma 4 for $G_3(L, k_{sib}, b_i)$ structure itself.

procedure TREE-PATH-G3-A();
begin
    TREE-PATH-G2();
    for \( l := 1 \) to \( L - 1 \) do
        for \( i := 1 \) to \( N_L \) do
            connect \( T_{i,v} \) as a parent to other PEs in the same sibling set;
            \( \{ \forall 2 \leq y \leq b_i \} \)
    end; \{ TREE-PATH-G3-A \}

procedure TREE-PATH-G3-B();
begin
    TREE-PATH-G3-A();
    for \( l := 1 \) to \( L - 1 \) do
        for \( i := 1 \) to \( N_L \) do
            connect \( T_{i,x} \) as a parent to other spare PEs in the same sibling set;
        for \( l := 1 \) to \( L - 2 \) do
            for \( i := 1 \) to \( N_L \) do
                connect \( T_{i,v} \) as a parent to a spare PE in grandchild blocks of \( B_i \);
                \( \{ \forall 1 \leq x \leq b_i, 2 \leq y \leq b_i \} \)
        end; \{ TREE-PATH-G3-B \}

5.4 Reconfiguration of \( d \)-ary trees

The reconfiguration algorithms for general trees are similar to those discussed in Chapter 3. However, we generalize the processes of recovery within a block and
out of a block as the following procedures. RECOVERY-INBLOCK corresponds to reconfiguration within a block. RECOVERY-OUTBLOCK is used for reconfiguration out of a block. For example, base is \( N_{i-1} \) and range is \( n_i \) for \( G_1 \); where base is \( n_{sib-1} \) and range is \( d \) for \( G_2 \). BJ-RESCUE-BI is similar to function as in RECOVERY-INBLOCK except that the spare PEs are from other blocks.

```plaintext
procedure RECOVERY-INBLOCK();
begins
    cut off every up and down paths of \( T_{i,1} \); mark \( B_i \) fail;
    if \( b_i \geq 2 \) then;
        for \( j := 2 \) to \( b_i \) do
            if \( T_{i,j} \) is good then begin
                if \( j < b_i \) then mark \( B_i \) hasspare else mark \( B_i \) nospare;
                \( T_{i,1} \) substituted by \( T_{i,j} \); mark \( T_{i,j} \) nogood; break; { for j loop }
            end;
        else mark \( T_{i,j} \) fail, cut off every up and down paths of \( T_{i,j} \);
end; { RECOVERY-INBLOCK }

procedure RECOVERY-OUTBLOCK(base, range);
begins
    mark system fail;
    for \( j := base + 1 \) to \( base + range \) do
        if \( B_j \) hasspare then BJ-RESCUE-BI();
        if system good then break; { for j loop }
end; { RECOVERY-OUTBLOCK }
```
procedure BJ-RESCUE-BI();
begin
  for k := 2 to bj do
    if $T_{j,k}$ good then begin
      if $k < bj$ then mark $B_j$ has spare else mark $B_j$ no spare;
      $T_{i,1}$ substituted by $T_{j,k}$; mark $T_{j,k}$ nogood;
      mark system good; break; { for k loop }
    end;
end; { BJ-RESCUE-BI }
The presented fault tolerant tree architectures not only are suitable for VLSI implementation but also have a higher system reliability and yield than other approaches. In this section, we only evaluate the performances of our $G_2(L, k_{sib}, b_i)$ and $G_3(L, k_{sib}, b_i)$ architectures due to the long links appear in $G_1$. We assume that the probability of link and switch faults is negligible compared with that of PE faults as assumed in all other fault tolerance designs [13]. This assumption is reasonable for sophisticated PE systems and for modeling link and switch faults into the corresponding PEs or as coverage failures [13]. We also compare the results with those of the previous fault tolerant binary tree design by Singh [13]. The architecture based on $G_2(L, k_{sib}, b_i)$ is $G_2(L, 2, 2)$ with $k_{sib} = 2$ and $b_i = 2$. $G_3(L, 2, 2$ or 1) implemented in Figure 3.6 is for $G_3(d, L, k_{sib}, b_i)$ architecture where $k_{sib} = 1$ and $b_i = 2$ or 1 depending on the number $i$. The examples use the reconfiguration strategies presented in Chapter 4.2 to perform the fault recovery and compute the reliability and yield enhancement.
Let $r = e^{-\lambda t} = e^{-0.01t}$ be the reliability of each PE, $c = \text{coverage coefficient}$ which is the probability that a non-faulty PE successfully replaces the faulty one through the reconfiguration algorithm, $L = \text{the number of levels in the tree}$. The reliability of $G_2(L, 2, 2)$ is straightforward:

$$R_{G_2} = [r^2 + r(1 - r)(1 + c)]^2 + 2r^3(1 - r)(1 + c) + r^2(1 - r)^2(1 + 2c + c^2)]^{L-1}$$

For $G_3(2, L, 1, 2 \text{ or } 1)$ architecture, each sibling set of lower level $l$ can be one of the following three states:

- State $p_l$: has spare nodes for level $l - 1$.
- State $q_l$: has no spares for level $l - 1$, but not failed.
- State $f_l$: failed.

For a sibling set at level $l = L$:

$$p_L = r^3$$
$$q_L = r^2(1 - r)(1 + 2c)$$
$$f_L = 1 - p_L - q_L.$$ 

For a sibling set at any other level such that $2 \leq l - 1 \leq L - 1$:

$$p_{l-1} = r^3(p_l + q_l)^2 + 3cr^2(1 - r)(p_l^2 + q_l) + 3c^2r(1 - r)^2p_l^2$$

$$q_{l-1} = r^2(1 - r)[3cq_l^2 + 3cp_lq_l + (1 - c)(p_l + q_l)^2] + r(1 - r)^2[4c^2p_lq_l + c(2 - c)p_l(p_l + q_l)]$$

$$+(1 - r)^3c^3p_l^2$$

$$f_L = 1 - p_L - q_L.$$
Therefore, the reliability of the example $G_3$ is:

$$R_{G3} = [r^2 + r(1 - r)(1 + c)][p_2 + q_2] + (1 - r)^2c^2p_2.$$ 

And the reliability of Singh's model should be:

$$R_{sys} = p_1^{(1)}[R_p + (1 - R_pc)] + R_p p_2^{(1)}$$

instead of:

$$R_{sys} = p_1 + R_p p_2$$

derived in [13]. The results and comparisons of different values of $c$ in a tree of level 8 are shown in Figure 6.1. It illustrates that both $G_2$ and $G_3$ have their system reliability better than that of Singh's design. This is because of the contribution of more redundant PEs in $G_2$ and $G_3$, although $G_3$ has only one more spare PEs at the root than Singh's design. In order to have a more meaningful comparison of the effectiveness of the two redundant designs, we use the normalized reliability improvement factor (NRIF) which takes the level of redundancy into consideration [13], where

$$NRIF = \frac{RIF \times \# \text{ of PEs in nonredundant tree}}{\# \text{ of PEs in the tree with redundancy}}$$

$$RIF = \frac{1 - R_{nr}}{1 - R_{sys}}$$

$R_{nr}$ = the reliability of the nonredundant tree

This factor scales down the RIF in proportion to the amount of hardware in each redundant design to normalize hardware losts. In Figure 6.2(a) with the perfect coverage factor ($c = 1$), the NRIF of $G_2$ is much larger than those of $G_3$ and Singh's design, where $G_3$ has a slightly better NRIF than Singh's design. This is also due to the additional spare PEs. At $c = 0.98$ as in Figure 6.2(b), the NRIF of $G_2$ is less
than those of $G_3$ and Singh's design in the beginning because the reconfiguration strategy does not allow spare PEs in $G_2$ to tolerate faults across levels. Yet, $G_2$ still has a better NRIF as time goes by (greater than 1.0 seconds) due to every tree PE has its spare PE respectively. The NRIF of $G_3$ is always slightly larger than that of Singh's design as expected.

The redundancy can be used to increase the overall yield for a large VLSI tree architecture following fabrication. For the yield enhancement evaluations, as in [13], if we let $r = \text{PE survival probability}$ (the probability that a PE is fault free following manufacture) and $c = \text{coverage coefficient}$ (the probability that all needed switches can be reliably programmed), the equation of the tree yield will be the same as that equation in reliability analysis. However, the tree yield does not reflect the true silicon yield because it is not normalized with respect to the extra area required by the redundant designs. To make a more reasonable comparison, the Chip Area Utilization Factor (CAUF) measuring the real utilization of a VLSI chip is used, where

$$CAUF = \frac{\text{(tree yield)} \times \text{(number of PEs in the expected nonredundant tree)}}{\text{(number of PEs used in the redundant tree)}}$$

This factor measures the fraction of fault free PEs that are actually utilized (on the average) in implementing the array on a chip. Clearly it measures the efficiency of silicon utilization. The results of yield versus PE survival probability ($r$) and CAUF versus $r$ at $c = 1, L = 8$ are plotted in Figure 6.3 and Figure 6.4 respectively. In Figure 6.3, $G_2$ has a better yield than $G_3$ and $G_3$ has a better yield than Singh's design. This is because of the contribution of more redundant PEs in $G_2$ and $G_3$. In Figure 6.4 for CAUF, $G_3$ and Singh's design is better than $G_2$ if PE survival
Figure 6.1: Reliability comparison of eight level binary trees.
Figure 6.2: NRIF comparison of eight level binary trees.

(a) $I = 8, c = 1.0$

(b) $I = 8, c = 0.98$
probability is high, such as larger than about 0.95; otherwise, $G_2$ is better, while $G_3$ always has a better CAUF than Singh's design. This is due to the extra spare PEs are wasted for a larger PE survival probability where only a few failures occur.
Figure 6.3: Yield comparison of eight level binary trees.

(a) $l = 8, c = 1.0$

(b) $l = 8, c = 0.95$
Figure 6.1: CAUF comparison of eight level binary trees.
CHAPTER 7
DISCUSSION AND CONCLUSION

All $G_1$, $G_2$ and $G_3$ architectures are evolved from the same block-oriented philosophy. The differences are additional links added into individual architecture to achieve the fault tolerance. We can consider $G_1$ architecture as a goal which has the best utilization of all spares at the same level but introduces some long links. The architectures $G_2$ and $G_3$ are introduced to provide the trade-offs between the performance and the complexity of VLSI layout and reconfiguration algorithms. For those who can afford a few number of spares but require an efficient VLSI layout, $G_2$ architecture should be the choice. Yet, for those who wish not to spend so many complex PEs but accept some digestible extra links, $G_3$ is a suitable architecture. It is also possible to combine levels from $G_1$, $G_2$ or $G_3$ together in the tree system. In addition, the number of spare PEs in $G_3$ can be reduced because $G_3$ can tolerate faults across levels, especially for the implementations in Figure 3.6(b) where spares can tolerate faults at level $l + 1$ and level $l - 1$. Then, as $G_3$ to $G_2$, if we adds into $G_1$ the corresponding extra links, $G_1$ can also extend the reconfiguration across one level. $G_1$ and $G_3$ can tolerate faults across more levels if more additional links are
added between the PEs in $B_i$ at level $l$ and the spare PEs in descendant blocks of $B_i$ at level $l+3, l+4, \ldots$. But we do not recommend these undesirable structures because they introduce the long link difficulty which is not practical for VLSI embedding.

Now let us look at the previous fault tolerant tree schemes proposed by other researchers. For the scheme by Raghvendra, Avizienis and Ercegovac [11], it can be derived from our binary $G_1$ architecture by two steps: 1) placing the only spare PE at each level $l$ in block $B_{N_{L-1}+1}$, $1 \leq l \leq L$; 2) adding the extra links required in their model at respective positions. Since this was not a practical design, they also introduced some extensions of this scheme which are still able to be derived from our binary $G_2$ architecture. Taking the one plotted in Fig.5 of their paper for example, it was obtained by similar steps: 1) placing the spare PE per sibling set $sib$ into the first block $B_{n_{sib-1}+1}$ of each sibling set; 2) adding the very little number of extra links into the individual position. It should be noted that their extension schemes not only have worse VLSI layouts but also have lower reliabilities and yield enhancements comparing with our $G_3$ tree architecture. This is because their extra links are almost all long links and the spares can not recover the faults across levels.

For the modular approaches, since they have many similarities and Singh [14] had proved his model is better than the one by Hassan and Agarwal [12], we only compare ours with the scheme by Singh which is actually an instance of our $G_3$ architecture implemented in Figure 3.6(a). The reduction has three steps: 1) taking the spare PEs out of the root; 2) placing the spare PE in each sibling set at level
l to be the spare in the correspondent module between level \( l - 1 \) and level \( l \) as in his modular scheme; 3) changing the links and switches between PEs in \( B_i \) at level \( l \) and spare PEs in the respective grandchild blocks of \( B_i \) at level \( l + 2 \) to his link and switch structures. However, our merits are the compact tree architecture for VLSI layout, having better reliability and yield enhancement (discussed in the last section), and being able to tolerate the faults through different reconfiguration algorithms as mentioned in section 3.3.

For the SOFT reconfigurable tree design by Lowrie and Fuchs [14], the adding of spares is limited and a faulty PE at levels near the root requires a lot of reconfiguration processing. It can be shown that our \( G_2 \) and \( G_3 \) architectures have more merits including higher reliabilities and yield enhancements, easy VLSI layouts and efficient reconfiguration algorithms. Finally, for the design by Dutt and Hayes [16], the \( G[k,T_N(d,l)] \) of NST's in their paper is just similar to our extension model of \( G_1(d,L,k_i,b_i) \). In this case, every \( k_i \) is set to have the same value and the spare PEs are nearly equally distributed into every block at the same level \( l \). Our design, however, dose not need the complex covering graphs and complicated reconfiguration algorithms to achieve the same performance in addition to having the advantage of being compacted into a rigid structure for the applicable VLSI layout. Also, their IIST's design is just as the extension of \( G_1 \) discussed in the beginning of this section.

In this thesis we have studied both categories of fault tolerant approaches for tree architectures. For approaches with performance degradation, algorithms to construct the tree and to find a shortest communication path between any cou-
ple of nodes are presented. For approaches in the second category, we introduced a novel block-oriented reconfiguration technique to the fault tolerant design of binary tree architectures for VLSI systems. It evolves three alternatives, $G_1(L, k_l, b_i)$, $G_2(L, k_{sib}, b_i)$ and $G_3(L, k_{sib}, b_i)$, for applications with different reliability and redundancy requirements. The effectiveness of this technique is due to the factor that sharing of spares in a level or among adjacent levels is allowed. These designs are extensible to general $d$-ary trees with a different number of PEs in each block, which are specified as $G_1(d, L, k_l, b_i)$, $G_2(d, L, k_{sib}, b_i)$ and $G_3(d, L, k_{sib}, b_i)$. We also evaluate the reliability and yield enhancement for $G_2(L, 2, 2)$ and $G_3(L, 1, 2$ or 1) and compare the results with those of Singh's fault tolerant scheme [13]. This emphasizes the advantages of our design.
APPENDIX A

AN IMPLEMENTATION OF SHORTEST PATH ALGORITHMS
This program is to decide the shortest path communication between any two nodes \((i, j)\) in a complete full ring binary tree structure, for both fault-free and faulty cases. The size \((n)\) of the tree is decided each time the level \((l)\) of the tree is read in. It uses the function, malloc, to generate tables depending on \(n\). Then it initializes the conditions (fn. initial), computes the best route of the fault free case (table). After reading in the faulty nodes (may have no faulty nodes), it updates the conditions (fault), then searches the shortest path for any two nodes (search) and prints out the path (show) repeatedly.

```c
#include <stdio.h>
#include <math.h>

#define good 1
#define bad 0
#define yes 0
#define no 1
#define cal(x,y) ((int) pow((double)x, (double)y))
#define level(x) ((int) floor(log10((double) x)/log10(2.0)))
#define pap(x) ((int) floor((double) x / 2.0))
#define listspace ((list_ptr) malloc(sizeof(list_type)))

int l;        /* # of levels */
int n;        /* # of nodes */
int leftleaf; /* for convenience */
int *f;       /* remember the faulty nodes */
int f_num;    /* # of faulty nodes */
int *path;    /* keep the path for shortpath */
int p_num;    /* keep the # of nodes for shortpath fn. */

typedef struct status {
```
int p;       /* condition flag to pap */
int bl;      /* condition to left brother */
int br;      /* condition to right brother */
int cl;      /* condition to left child */
int cr;      /* condition to right child */
int pre;     /* the predecessor for the shortpath fn. */
int length;  /* the length condition for the shortpath fn. */
int visit;   /* the visit flag for the shortpath fn. */
} node_type, *node_ptr;

typedef struct list {
    int id;       /* for matrix table route */
    struct list *next;
} list_type, *list_ptr;

node_ptr node;       /* for node, skip 0 for convenience */
list_type **route;    /* for route */

/*********************************************************/
/*
main()
{
    int i = f_num = 0;
    int source, dest;
    char *malloc();
    void nospace(), initial(), table(), fault(), shortpath(), show();
    int change();

    while (scanf("%d\n", &l) != EOF)
    {
        n = cal(2,1) - 1;
        leftleaf = cal(2,1-1);

        if ((node = (node_ptr)malloc(sizeof(node_type)*(n + 1))) == NULL)
            nospace();
        if ((f = (int *) malloc(sizeof(int) * n)) == NULL)
nospace();
if ((path = (int *) malloc(sizeof(int) * n)) == NULL)
    nospace();

initial();
table();

printf("\n\n level = %d, \ of nodes = %d\n\n", l, n);
printf(" faulty nodes => ");

while (scanf("%d", &f[i]) != 0)
{
    if (f[i] != bad)
    {
        printf("%d", f[i]);
        fault(f[f_num = i++]);
    }
    else
    {
        while (scanf("%d %d\n", &source, &dest) == 2)
            if (source == bad || dest == bad)
                break; /* while loop of source and dest */
        else
            {
                if (change(source, dest) == good)
                    shortpath(source, dest);
                show(source, dest);
            }
        break; /* while loop of f[i] */
    }
}

exit(0);

/******************************************************************************
 nospace() -- nospace message of the calling of malloc()
*******************************************************************************/
void nospace()
{
    printf("\n\n level = %d, # of nodes = %d\n\n", l, n);
    printf(" ****** cannot create space ******\n");
    exit(1);
}

/*
initial() -- initialize the node conditions, based on the complete full ring binary tree structure.
***********************************************************************/

void initial()
{
    int i, j;

    for (i = 0; i <= n; i++)
    {
        node[i].p = node[i].bl = node[i].br = good;
        node[i].cl = node[i].cr = good;
        node[i].pre = node[i].visit = no;
        node[i].length = n;
    }

    /* root */
    node[l].p = node[l].bl = node[l].br = bad;
    /* leaves, i.e., level = l */
    for (i = leftleaf; i <= n; i++)
        node[i].cl = node[i].cr = bad;
    /* left and right sides */
    for (j = 1; j <= l; j++)
        node[i = cal(2,j)].bl = node[i-i].br = bad;
}
/***************************************************************/
/* table() — The fault free route of each pair of (i,j), based on the */
/* complete full ring binary tree structure. The route is a pointer */
/* to an array (n+1) of pointer to a matrix (n+1)*(n+1) of list_type, */
/* such that route[i][j] would be the matrix[i][j]. */
/* The algorithm is to call shortpath to compute the fault free */
/* route[i][j], for any i > j, shortpath will call reverse for i < j. */
/* ***************************************************************/

void table()
{
    int i, j, k;
    int l_i, l_j;
    char *malloc();
    void nospace(), shortpath();

    /* skip index 0 for convenience */
    route = (list_type **) malloc(sizeof(list_type *) * (n+1));
    *route = (list_type *) malloc(sizeof(list_type) * (n+1) * (n+1));

    if (route == NULL || *route == NULL)
      nospace();

    for (i = 1; i <= n; i++)
      route[i] = *route + i * (n+1);

    for (i = 1; i <= n; i++)
      for (j = i + 1; j <= n; j++)
        shortpath(i,j);
}
void fault(k)
int k;
{
    int i;
    int k_p;
    void freespace();

    node[k].p = node[k].bl = node[k].br = bad;
    node[k].cl = node[k].cr = bad;
    node[k].visit = yes;

    if (k < leftleaf)
        node[2*k].p = node[2*k + 1].p = bad;
    if (k > 1)
    {
        node[k-1].br = bad;
        if (k == 2 * (k_p = pap(k)))
            node[k_p].cl = bad;
        else
            node[k_p].cr = bad;
    }
    if (k < n)
        node[k+1].bl = bad;

    for (i = 1; i <= n; i++)
    {
/*
  fault() -- updates the node conditions due to the faulty node k, and
  calls freespace to empty route[i][k] and route[k][i] and reports
  bad of their id for any i. In fact, the only condition of node[k]
  to be concerned about is node[k].visit = yes, which prevents the
  contributions of all other condition flags in shortpath. For the
  convenience in shortpath, however, we update all the informations.

  **********************************************************************
  fault0 — updates the node conditions due to the faulty node k, and
  calls freespace to empty route[i][k] and route[k][i] and reports
  bad of their id for any i. In fact, the only condition of node[k]
  to be concerned about is node[k].visit = yes, which prevents the
  contributions of all other condition flags in shortpath. For the
  convenience in shortpath, however, we update all the informations.
  **********************************************************************
*/
```c
freespace(i, k);
freespace(k, i);
}

int change(int i, j)
{
    int k;
    list_ptr p = &route[i][j];

    if (p->next == NULL && p->id == i)
        return good;

    while ((p = p->next) != NULL)
        for (k = 0; k <= f_num; k++)
            if (p->id == f[k])
                {freespace(i, j);
freespace(j, i);
                return good;
            }

    return bad;
}
```

/***************************************************************************
change() -- find out whether the faulty nodes affect the existing route[i][j]. This was considered in fault for either of i or j itself being faulty. In change, it only checks with other faulty nodes which this route passes through. It calls freespace to empty the existing route[i][j] and route[j][i] and reports good for shortpath to find another shortest path.
***************************************************************************
*/
freespace() -- free the space of route[i][j] for future use. The first node of route[i][j] is to be kept and reports bad.

*******************************************************************************/

void freespace(i, j)
int i, j;
{
    list_ptr p = &route[i][j], temp;
    void free();

    p->id = bad;
    while ((temp = p->next) != NULL)
    {
        p->next = temp->next;
        free(temp);
    }
}

*******************************************************************************/

shortpath() -- Use an algorithm to find out the shortest path between any two nodes based on the new internetworking binary tree structure. The algorithm is similar to the one by Dijkstra E.W. in "A note on two problems in connexion with graphs", Numerische Mathematik, vol.1, P269-271, Oct 1959.

It calls do.pre to compute the possible shortest paths from i to a connected but good node, k. If no one is found (stop), it stops the searching and reports failure, else it selects the next visited node. This procedure will continue until a shortest path is found or a failure is reported. If a path is found, it calls create and reverse to produce route[i][j] and route[j][i] respectively. This algorithm guarantees to find one shortest path from existing ones.

*******************************************************************************/

void shortpath(i, j)
int i, j;
{

int h, k, min, stop = 0, num = 0;
void do_pre(), create(), decide(), reverse();
int compare();

for (h = 1; h <= n; h++)
{
    node[h].pre = no;
    node[h].length = n;
    node[h].visit = no;
}

node[i].length = 0;
node[i].visit = yes;
k = i;

do
{
    min = n;

    do_pre(pap(k), k, node[k].p);
    do_pre(k+1, k, node[k].br);
    do_pre(2*k, k, node[k].cl);
    do_pre(2*k+1, k, node[k].cr);

    /* Find out the next visited node based on the shortest
    paths and node conditions of them. */
    for (h = 1; h <= n; h++)
        if (node[h].visit == no && node[h].length < min)
            min = node[k = h].length;

    if (k != stop)
        node[k].visit = yes;
    else
        freespace(k, j);

} while ((stop = k) != j & route[k][j].id != bad);

h = k;
if (route[k][j].id != bad)
  while (h != i)
  {
    path[num++] = h;
    h = node[h].pre;
  }

if ((p_num = num) > 0)
{
  route[i][j].id = i;
  create(i, j);
  route[j][i].id = j;
  reverse(i, j, num-1);
}

******************************************************************************
do_pre() — Find out the possible shortest paths from i to node k which is connected but good.
*******************************************************************************/
void do_pre(h, k, cond)
int h, k, cond;
{
  /* it may have a long path */
  if (h >= 1 && h <= n)
    if (node[h].visit == no && cond == good)
      if (node[k].length + 1 < node[h].length)
        {
          node[h].pre = k;
          node[h].length = node[k].length + 1;
        }

/******************************************************************************
create() -- Create the route[i][j] based on the remembered node #'s of the shortest path from i to j by shortpath.
**********************************************************************
*/

void create(i, j)
int i, j;
{
    list_ptr p = &route[i][j], new;
    char *malloc();
    void nospace();

    while (p_num >= 0)
    {
        if ((new = listspace) == NULL)
            nospace();
        p->next = new;
        new->next = NULL;
        new->id = path[p_num];
        p = new;
    }
}

/**********************************************************************
reverse() -- Create the route[j][i] based on the remembered node #'s of the shortest path from i to j by shortpath.
**********************************************************************
*/

void reverse(i, j, num)
int i, j, num;
{
    list_ptr p = &route[j][i], new;
    char *malloc();
    void nospace();

    while (p_num++ < num)
    {

if ((new = listspace) == NULL)  
    nospace();  
    p->next = new;  
    new->next = NULL;  
    new->id = path[p_num+1];  
    p = new;  
}

new->id = i;

/***************************************************************************/
** show() — If route[i][j] exists, print it out; otherwise print the     **
** failure message.                                                        **
/***************************************************************************/
*/

void show(i, j)
int i, j;
{
    int h = 0;
    list_ptr p = &route[i][j];

    printf("\n\n route[%d][%d] = ", i, j);

    do
        if (h < 6)
            {
                printf("%d --> ", p->id);
                ++h;
            }
        else
            {
                printf("\n\n\t%d --> ", p->id);
                h = 1;
            }
    while ((p = p->next) != NULL);
if (route[i][j].id == bad)
    printf("failure\n\n");
else
    printf("success\n\n");

/**************************************************************************
THE END
**************************************************************************/
REFERENCES


