SCISPACE
formerly Typeset

Open access • Journal Article • DOI:10.1109/TC.1987.1676858

Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance

— Source link

Lowrie, Fuchs

Institutions: University of Illinois at Urbana–Champaign

Published on: 01 Oct 1987 - IEEE Transactions on Computers (IEEE)

Topics: Tree (data structure), Binary tree, Fault tolerance and Redundancy (engineering)

Related papers:

- Fault Tolerance in Binary Tree Architectures
- A Graph Model for Fault-Tolerant Computing Systems
- The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors
- A Fault-Tolerant Modular Architecture for Binary Trees
- An optimal 2-FT realization of binary symmetric hierarchical tree systems

Share this paper:

View more about this paper here: https://typeset.io/papers/reconfigurable-tree-architectures-using-subtree-oriented-5d4mpun1ue

March 1987

UILU-ENG-87-2216
CSG-65

COORDINATED SCIENCE LABORATORY
College of Engineering

RECONFIGURABLE TREE ARCHITECTURES USING SUBTREE ORIENTED FAULT TOLERANCE

Matthew B. Lowrie

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-87-2216 (CSG-65)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center Hampton VA 23665
8a. NAME OF FUNDING / SPONSORING ORGANIZATION NASA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG 1-613
8c. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center Hampton, VA 23665	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance

12. PERSONAL AUTHOR(S) Lowrie, Matthew B.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ____ TO ____	March 1987	46

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
VLSI, binary tree, N-ary, fault tolerance, tree architecture
Subtree Oriented Fault Tolerance (SOFT)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

An approach to the design of reconfigurable tree architectures is presented in which spare processores are allocated at the leaves. The approach is unique in that spares are associated with subtrees and sharing of spares between these subtrees can occur. The Subtree Oriented Fault Tolerance (SOFT) approach is more reliable than previous approaches capable of tolerating link and switch failures for both single chip and multi-chip tree implementations while reducing redundancy in terms of both spare processors and links. VLSI layout is O(n) for binary trees and is directly extensible to N-ary trees and fault tolerance through performance degradation.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

RECONFIGURABLE TREE ARCHITECTURES USING SUBTREE ORIENTED FAULT TOLERANCE

Matthew B. Lowrie

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801

Abstract- An approach to the design of reconfigurable tree architectures is presented in which spare processors are allocated at the leaves. The approach is unique in that spares are associated with subtrees and sharing of spares between these subtrees can occur. The Subtree Oriented Fault Tolerance (SOFT) approach is more reliable than previous approaches capable of tolerating link and switch failures for both single chip and multi-chip tree implementations while reducing redundancy in terms of both spare processors and links. VLSI layout is $O(n)$ for binary trees and is directly extensible to N-ary trees and fault tolerance through performance degradation.

This research was supported in part by the Microelectronics and Computer Technology Corporation (MCC) under a VLSI/CAD grant, by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613, and by a fellowship from the A.T.&T. Bell Laboratories.

I. INTRODUCTION

A continually rising demand for high performance computation has created a need for highly concurrent computer architectures. One architecture which has received significant attention is the tree topology [1, 2, 3, 4]. Tree architectures have an inherent ability to compute concurrently with typical communication times between the n processors being $O(\log n)$. However, as the number of processor nodes and communication links increases, the probability of single or multiple failures within structured concurrent architectures becomes unacceptably large. Consequently, recent interest has arisen in designing the ability to reconfigure concurrent architectures with one or more faults. Reconfigurability, which is one aspect of fault tolerance, is especially significant in tightly coupled tree architectures where the failure of a single link or processor can result in the subsequent loss of all communication with processors in the subtree below the faulty element.

One of the initial reconfigurable binary tree proposal was made by Hayes, who developed a procedure for constructing 'optimal' 1-fault tolerant trees [5], which has since been extended by Kwan and Toida [6]. Raghavendra, Avizienis, and Ercegovac (RAE) [7], improved on these proposals by adding sufficient redundant lines in order to tolerate multiple failures. Link redundancy for binary trees in the RAE approach is as high as 200%, and VLSI layout may require $O(n \log n)$ area [8]. Hassan and Agarwal [8], recently presented a modular technique which allocates one spare to multilevel groups of processors. This scheme is conceptually similar to the RAE approach in that it dedicates every spare to one specific group of processors, but has the advantage of $O(n)$ layout and modularity for multichip architectures. Another proposal for reconfiguration which is applicable to tree architectures has been proposed by Rosenberg [9, 10]. The approach requires a collinear layout with each node requiring access to a $\log n$ bus. Redundancy in terms of switching transistors is $O(\log n)$ for each node. The switching structure provides for efficient utilization of spare processors. However, fault tolerance for the communication lines and the switching transistors is not considered.

One of the important objectives in designing for reconfiguration is efficient utilization of spares. If the architecture has k spare processors then the objective is to be able to tolerate any combination of k processor failures through reconfiguration. This should be accomplished with a reasonable increase in interconnect, manageable layout complexity for large numbers of processors, and a bounded number of pins per chip. In addition, failures in interconnect and switching structures should also be tolerable through reconfiguration.

A strategy for satisfying these objectives for binary tree architectures is presented in this paper. The approach places spare processors at the leaves of the tree and provides for considerable flexibility in reconfiguration through sharing of spares between adjacent subtrees. This strategy, which is referred to as Subtree Oriented Fault Tolerance (SOFT), utilizes a virtual displacement technique to reconfigure a spare processor into the tree. The capability of sharing spare processors between subtrees provides the SOFT approach with significantly higher reliability than previous techniques allowing for switch and link fault failures, where reliability is the probability that the tree is functional at a time t, given that it was fault free at time 0. In contrast to other proposals, SOFT is able to tolerate link and switch failures while reducing the number of redundant links between processors. For binary trees, the approach is shown to yield $O(n)$ layout. The architecture can be partitioned on separate chips for arbitrarily large trees, while providing fault tolerance for both on and off-chip connections. Fault tolerance through performance degradation is also possible with a SOFT design, as well as application to N-ary trees.

In Section II of this paper, the SOFT architecture is presented for binary trees. Considerations for implementing a SOFT binary tree are discussed, including the placement of spare processors and communication links. Section III provides a formal analysis of reconfigurability in SOFT binary trees in the presence of processor, switch, and link failures. In Section IV, comparisons between the reliability of SOFT and past reconfigurable designs are presented. Finally, Section V extends the SOFT concept to N-ary trees.

II. THE SOFT DESIGN FOR BINARY TREES

The SOFT approach to reconfigurable tree architectures employs both spare processors at the leaves of the tree and additional links between processors to maintain a complete tree topology in the presence of multiple faulty processors and links. An example of SOFT architecture is illustrated in Figure 1. At levels high in the tree, failure of a node results in bypassing the node, thereby allowing information to flow directly between the faulty node's father and one of its sons. Thus, the faulty node's son assumes the tasks allocated to its failed father. Since the son is performing the tasks of its father, another processor must be found to assume the son's tasks. In a similar fashion, one of the son's sons assumes its responsibilities. This 'logical displacement' continues until a spare is configured in at the leaves. A detailed discussion and an example of SOFT reconfiguration are presented in Section III.

A. Terminology for Binary Trees

All trees are said to have $i+1$ levels. The root is at level 0 and the leaves are on level i. The term 'upper levels' refers to levels 0 through $i-1$. The root is labeled 1, the left child of any node n is labeled $2n$ and the right child is labeled $2n+1$. Two nodes are *adjacent* if they are connected by a nonredundant or redundant communication link. The *father* of a node n on level k (f_n) is the adjacent node on level $k-1$. Similarly, the son of a node n is son_n. f_l and son_l represent the father and son nodes of link l, respectively. The *brother* of a node n (b_n) is the single node having the same f_n. b_l refers to either node connected to a redundant link l. The *left-most descendant* of a subtree is the node which can be found by following only left descendants of the root of the subtree. Right-most descendants are defined similarly. The *cousin* of a node n $(cous_n)$ is the left(right)-most node on the same level if n is the right(left)-most son of the root. For all other n, $cous_n$ is $n-(+)1$ if n is a left(right) son of its father. The *ancestor* of a node n on a level q (A_n^q) is the single node on level q which contains n in its subtree.

B. Allocation of Redundant Nodes and Links

The number of spare processors supported by the SOFT architecture is 2^c, where c is an integer: $0 \leqslant c \leqslant i-1$. Algorithm 1 is an algorithm for positioning these spares. The redundant links required by the SOFT architecture are allocated as described by Algorithm 2. The SOFT binary tree of Figure 1 was generated by Algorithms 1 and 2 with $i=4$ and $c=2$. A subtree with leaves $x+k\,2^{i_{SST}}$ to $x+(k+1)2^{i_{SST}}-1$, where x is the left-most leaf of the root and $0 \leqslant k \leqslant 2^c-1$, is referred to as a Spare SubTree, or SST. Each SST has an *associated spare* which is adjacent to its right-most leaf. The spare adjacent to an SST's left-most leaf is referred to as its *nonassociated spare*. In contrast to X-tree or Hyper-tree structures [11, 12, 13], the SOFT topology is not a half-ring structure in which each level contains cousin connections instead of the n to b_n connections utilized by SOFT.

ALGORITHM 1: Placement of Spares.

Begin
- $i_{SST} := i - c$; {height of spare subtree}
- for k := 1 to $2^{i-i_{SST}}$ do begin
  - x := left-most leaf of root;
  - $n := x + k\,2^{i_{SST}} - 1$
  - add spare and connect as right son to n ; {associated spare of SST k}
  - connect spare to $cous_n$;
  - end
- end.

ALGORITHM 2: Placement of Redundant Links.

```
Procedure Brother_connect (n : node)
    begin
        connect n to b_n ;
        if level(n) < i        {n is not a leaf}
            then begin
                Brother_connect(left son of n);
                Brother_connect(left son of b_n);
                end
            else
                if not(adjacent_to_spare(n))
                    then connect n to cous_n ;
        end
begin
    Brother_connect(left son of root);
    end.
```

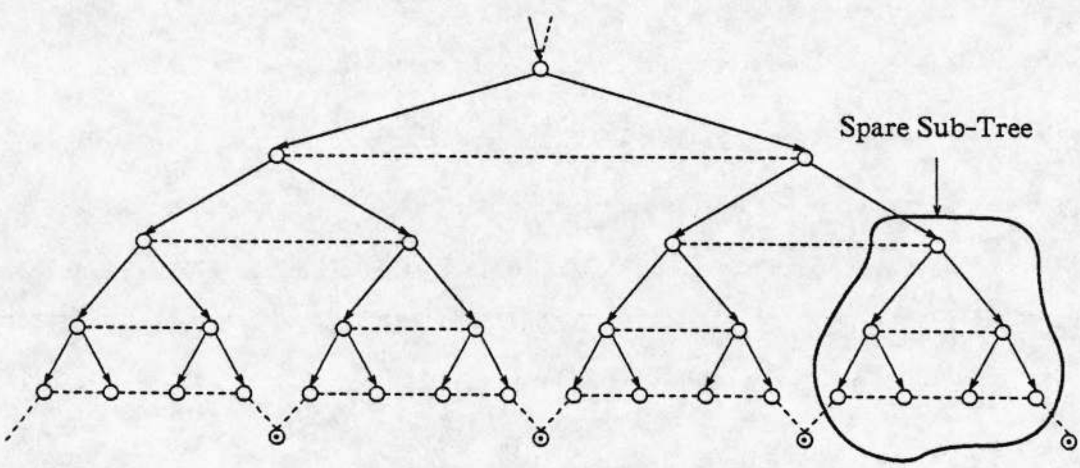


Figure 1. SOFT Architecture with $c = 2$ and $i = 4$.

C. *Redundancy Calculations*

The calculations for percentage of spare redundancy are straightforward:

$$\% \text{ node redundancy} = \frac{2^c}{2^{i+1}-1}.$$

The number of redundant links is

$$\text{redundant links} = \left(\sum_{g=0}^{g=i-2} 2^g \right) + 2^i + 2^c + 1$$

Thus, the percentage of redundancy for links is

$$\%\ \text{link redundancy} = \frac{2^{i-1} + 2^i + 2^c}{2^{i+1}-1}$$

For large i, this is approximately .5 + .25 + (% redundancy of spares). Table 1 enumerates the possible percentage node redundancies for large i and the corresponding percentage of link redundancy.

D. Implementing SOFT Architectures

Implementing SOFT architectures involves the following assumptions.

ASSUMPTION 1: Input/output through the leaves is not required. □

This is not a deficiency in the SOFT philosophy but a convenience in describing the architecture. The assumption is not unreasonable since many of the algorithms appropriate for tree architectures do not require such I/O [1, 2, 4]. In fact, the classical H-tree layout cannot accommodate I/O through the leaves for large trees.

Table 1. SOFT Redundancy.

% Node Redundancy	% Link Redundancy
≈25.0%	≈100%
≈12.5%	≈87.5%
≈6.25%	≈81.25%
$\frac{1}{2^{i+1}-1}$	≈75%

ASSUMPTION 2: All processors in the tree are identical. □

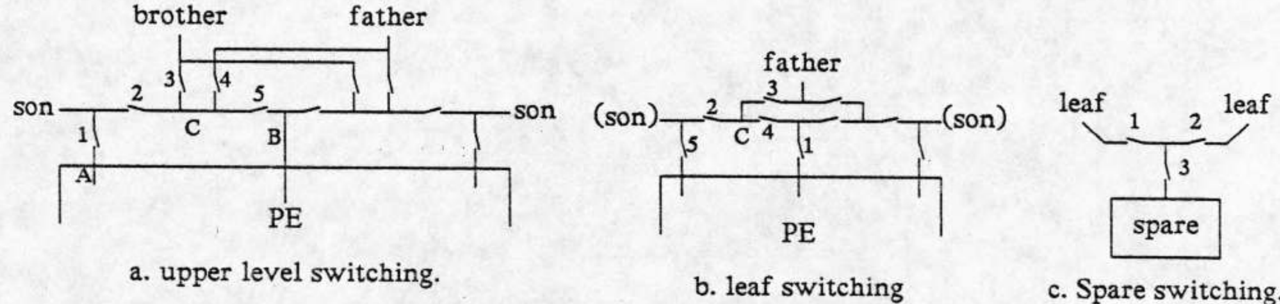
The necessity of this assumption is clear in light of the manner in which reconfiguration proceeds. The assumption may be relaxed somewhat since each processor on level l will only be asked to serve as a replacement for a processor on level $l-1$. Most well-known algorithms for tree architectures utilize identical processors [1, 2, 4].

1) Switching Scheme

The virtual processor displacement concept of SOFT reconfiguration can be implemented with the switching scheme of Figure 2. Since all switching due to reconfiguration is performed by these switches, design of the processing elements is independent of the reconfiguration scheme.

2) Multichip Trees

If an entire tree cannot fit onto a single chip or wafer, then the tree must be partitioned for chip allocation. The major consideration on dividing a tree into subpieces is imposed by pin limitations. Partitioning a SOFT tree is straightforward. From Figure 1, it can be seen that at most one additional link per chip is required for chips containing no leaf processors. At level i, more than one link must be added per chip. However, by observing that there are two redundant links per node at the leaves, which equates to the two father-to-son links of nodes at upper levels, it is evident that the number of pins per chip at the lower levels will be less for chips containing leaf



a. upper level switching. b. leaf switching c. Spare switching.

Figure 2. SOFT switching in binary trees.

processors. While the implication is that a SOFT tree will require at least two different kinds of chips, the same may be true for modular sparing approaches such as [7, 8], or even nonredundant trees if the tree has an odd number of levels.

3) VLSI Layout of SOFT Trees

To efficiently lay out a SOFT binary tree in VLSI, an adjustment in the general architecture is made in order to employ a variation of the optimal $O(n)$ H-tree layout [14, 15] in which the leaves are not fully connected. VLSI layout for a tree of $i \leqslant 4$ follows the layout of Figure 3a. The location of spares depends upon the percentage of spares allocated to the tree. As in Algorithm 1, the spares are located on the nonbrother redundant links. For trees with $i > 4$ the layout algorithm presented below results in optimal area of $O(n)$. The result of Algorithm 3 with $i = 7$ is depicted in Figure 3b. The ellipses represent the 5-level subtrees constructed in the first 'for loop' of the algorithm.

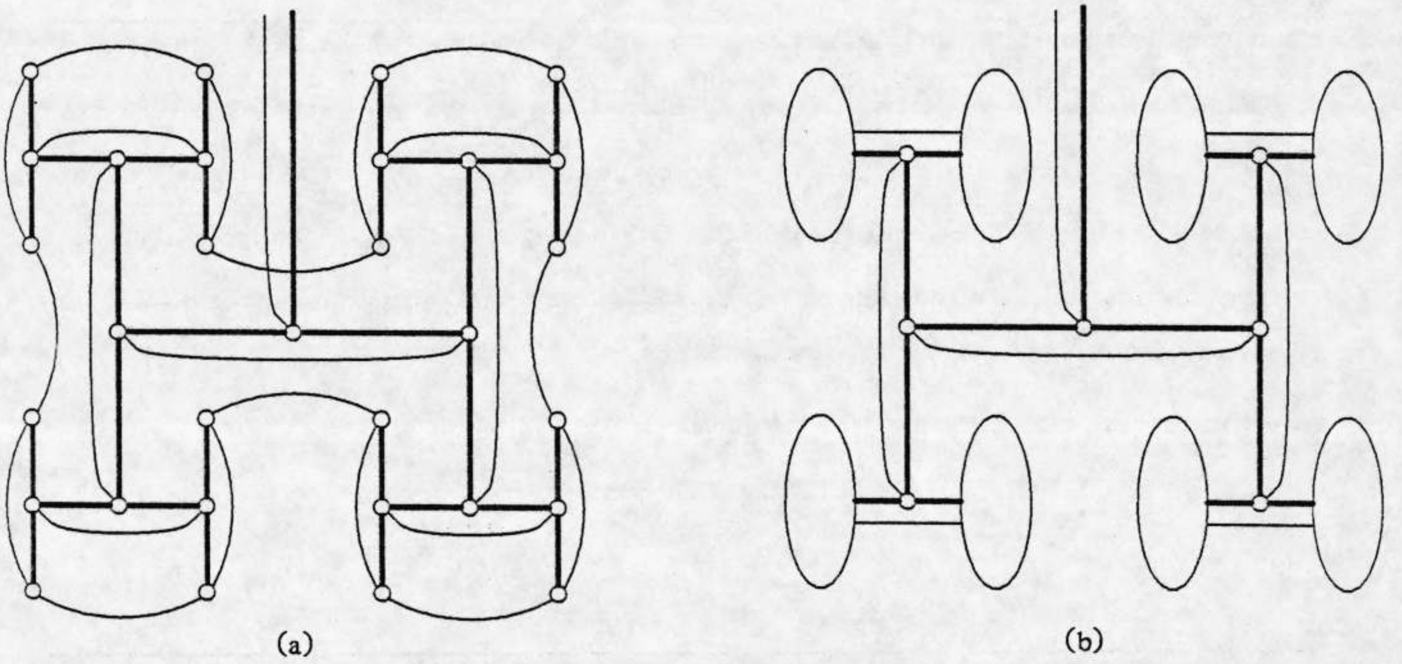


Figure 3. VLSI Layout of SOFT Architecture

THEOREM 1: The SOFT layout, as described in Algorithm 3, for a binary tree of n leaves, has $O(n)$ area.

PROOF: It is well known that the area of an H-tree is $O(n)$ [15]. The layout can be thought of as having $O(\sqrt{n})$ rows and columns. The layout produced by Algorithm 3 (as in Figure 3) has at most one redundant link (additional row or column) parallel to each row or column of the $O(n)$ H-tree layout. Since each spare can be thought of as lying in a redundant link, the edge of the square corresponding to the layout of a SOFT tree is at most $O(k\sqrt{n})$, where k is a constant which, in the worst case is between 2 and 3. Since this is still $O(\sqrt{n})$, the area of the SOFT layout is $O(n)$. □

The modification to the general SOFT approach necessary for VLSI layout has the following implications:

(1) While the maximum number of spares is ≈25%, the minimum is increased from an arbitrarily small percentage (1 per tree) to ≈3% (1 per 5 level subtree).

ALGORITHM 3: VLSI Layout of Trees with i>4.

```
begin
    for k := 1 to 2^(i-4) do
        z(k) := a 5-level SOFT tree constructed by Algorithms 1 and 2;
    for k := 1 to 2^(i-5) do
        connect root(z(2*k)) to root(2*k-1));
    construct main tree (m) as an i-5 level nonredundant binary tree;
    apply Algorithm 2 to m omitting all cousin connections;
    for k := 1 to 2^(i-5)
        x := left-most leaf of root;
        move to x+k-1
        connect z(2*k) as left son;
        connect z(2*k-1) as right son;
        end
    end.
```

---

(2) Since the 5 level subtrees do not share any connections among spares, SSTs cannot 'borrow' spares from neighbors not within that subtree. This does not, however, affect the reliability analysis presented in Section IV, which shows significant reliability enhancement is gained with the SOFT strategy.

III. FAULT TOLERANCE IN SOFT

Necessary and sufficient conditions are presented for reconfigurability in SOFT architectures. Reconfiguration for faulty processors is considered in Section A, followed by an analysis of reconfiguration for link failures and switch failures in Section B. The fault model for PEs and links is functional in nature and includes any fault affecting the correct operation of the processor or the link under consideration. The fault model for switches consists of stuck-open or stuck-closed faults. Algorithms for dynamic reconfiguration are presented in Section C. Finally, static reconfiguration is discussed in Section D.

A. Tolerance of Processor Failures

In Section 1, basic properties of SOFT reconfiguration are presented. Based on these properties, necessary and sufficient conditions for reconfigurability are derived in Section 2.

As discussed in Section II, failure of a node in an upper level results in a series of displacements until a spare is configured in. If a leaf node n fails and it is adjacent to its SST's spare, n is simply bypassed and replaced by the spare. If n is replacing f_n, n takes b_n as one son and the spare as its other son. If n is not adjacent to the SST's associated spare, it must be replaced by or take as its second son the non-brother leaf adjacent to it, a_n. If a son of f_{a_n} is adjacent to a spare, f_{a_n} may use that spare as a second son, otherwise it takes a leaf adjacent to one of its sons. This displacement continues along level i until a spare is configured in. By convention, if a failure occurs in an SST, it displaces toward its associated spare if possible. If there are two failures in the SST, it must configure in the nonassociated spare. If the nonassociated spare had been configured in to its associated SST, then this SST must take its nonassociated spare. This continues until an unused spare is configured in. In Figure 4, an example of reconfiguration with 4 faults and 4 spares is presented.

In analyzing the reliability of a reconfiguration scheme, it is necessary to determine both what fixed fault subsets (a set of processors in the tree designated as having failed) are reconfigurable

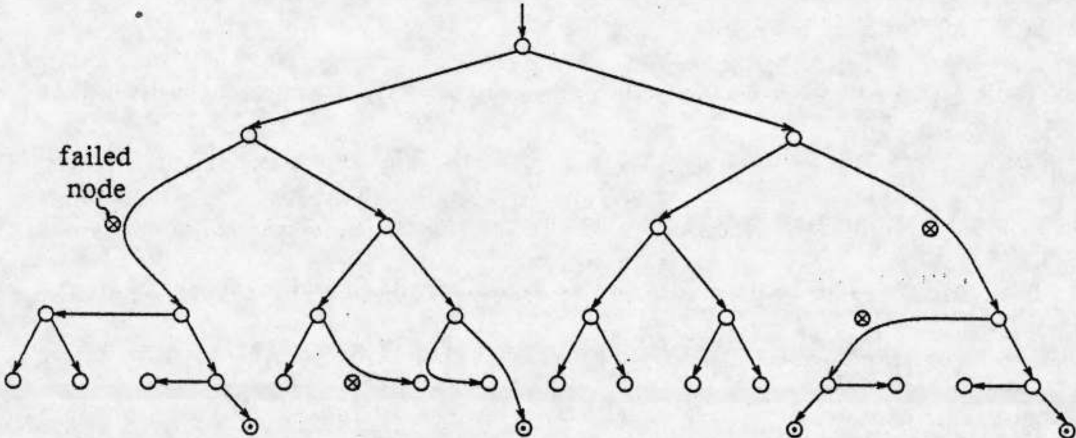


Figure 4. Reconfiguration in an i=4, c=2 tree with 4 faults.

(i.e., static reconfiguration), and for an existing machine with failures, whether the failure of an additional specific processor can be tolerated (i.e., dynamic reconfiguration). For SOFT dynamic reconfiguration, the order in which faults occur does not affect the ability of the architecture to reconfigure. As a result, if it is known exactly what sets of faults can be reconfigured, then a fault which occurs at time t can be reconfigured if and only if the new fault subset is reconfigurable. Similarly, if it is known exactly what failures can be reconfigured given a set of faults, then all sets of reconfigurable faults can be inductively determined. Consequently, the reconfigurability analysis of this section is applicable to both scenarios.

1) Properties of SOFT Reconfiguration

DEFINITION 1: *Displacement* is the logical movement of a node n to the physical position corresponding to f_n in order to replace f_n due to either the failure or displacement of f_n. At the leaves, displacement includes the logical movement of a leaf to the left or right in order to replace b_n or $cous_n$. Displacement through node n refers to the act of displacing n. □

DEFINITION 2: *Double displacement* refers to an attempt to displace a node twice. For example, if n is displaced, it is assuming f_n's tasks. If displacement were to occur through n again, n would assume g_n's tasks. □

The following Lemmas present basic aspects of reconfigurability which enable derivation of the necessary and sufficient conditions presented in Section 2.

LEMMA 1: Double displacement occurs at upper levels if and only if displacement is attempted through a faulty node.

PROOF: Displacement only occurs in descendants of faulty nodes. The lemma is not concerned with displacement of leaves so no double displacement occurs due to unavailability of spares. For two displacements to intersect at one node (double displacement, by Definition 2), the displacement due to a failure higher in the tree must intersect a previously displaced node. But it is necessary to pass through a father in order to reach one of its descendants. □

LEMMA 2: SOFT trees can not reconfigure using double displacement.

PROOF: By Definition 3, at upper levels some node n is assuming g_n's tasks. But n is not adjacent to b_{f_n}. Thus communication with b_{f_n} would be lost, and it would not be possible to maintain the rigid tree topology if double displacement occurred. At level i, the father of the double displaced node would be adjacent to only one leaf processor. □

LEMMA 3: At most one of n and b_n can be displaced.

PROOF: Displacement of both would imply that either f_n has been displaced twice, which is not possible by Lemma 2, or there is a displacement through f_n and f_n has failed, which is also impossible by Lemmas 1 and 2. □

LEMMA 4: Two failures within an SST, or displacement of the root of the SST, and a failure within the SST, are reconfigurable if and only if both spares adjacent to the SST can be reconfigured in as sons of leaves in the SST.

PROOF: Displacement of the root of the SST is the same as failure of the root of the SST in terms of reconfiguration below that level. Consequently, only failures within the SST need be considered. If only one adjacent spare can be configured into the SST, and there are two failures in the SST, then reconfiguration is not possible by Lemma 2. If both adjacent spares can be used, then

reconfiguration through fault free nodes is clearly possible, i.e., a fault subset such as Figure 5a cannot exist with only two failures. □

2) Analysis of Reconfigurability

In previous approaches, unreconfigurable multiple failures correspond to more than one fault in a group of nodes which have been allocated a single spare [7, 8]. Unreconfigurable multiple failures in the SOFT approach, regardless of the number and location of spares, correspond to faults which force double displacement. For example, consider the failure of a node n. Displacement must occur through a son_n. Consequently, the presence of a fault subset as depicted in Figure 5a is not reconfigurable. If *left* son_n has failed, then displacement must occur through *right* son_n which implies that one of *right* son_n's sons must be fault free for reconfiguration to occur. The failure of *left* son_n can be thought of as forcing the reconfiguration into the subtree with *right* son_n as its root. Thus, Figures 5b and 5c are not reconfigurable since there is no path from the highest faulty node to the leaves through only good nodes.

In order to determine what fault scenarios are reconfigurable, spare sufficiency (SS) is defined. SS is a boolean value associated with each node of the tree.

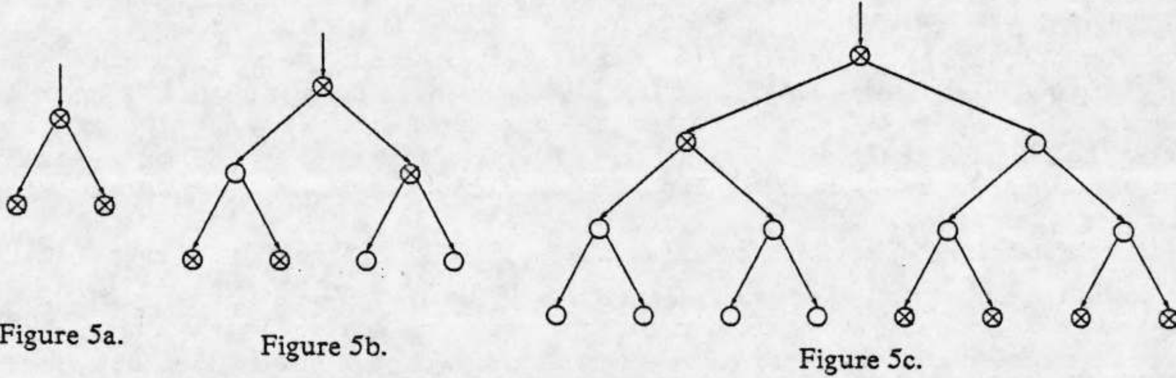


Figure 5. Unreconfigurable faults at upper levels.

DEFINITION 3: The SS of a spare s is 1 if and only if s is fault free and not configured into the array, or, the nonassociated spare of the SST has SS of 1 and is not configured into the array as a son of s's associated SST, or s is faulty, and the SST for which s is not associated has SS of 1. The SS of a node within an SST is equal to the SS value of its associated spare. The SS of a node n in a level above the SSTs is 1 if and only if $SS(left\ son_n)$ or $SS(right\ son_n)$ is 1 and n is fault free, where $SS(n)$ denotes the spare sufficiency of n. □

In addition, the *shiftability* of a displaced node n, denoted as $s(n)$, is formally defined as:

$$s(n) = SS(b_n) + \sum_{q=level(n)-1}^{q=1\ by\ -1} SS(b_{A_n^q}) \left[\prod_{l=level(n)-1}^{l=q\ by\ -1} good(A_n^q) \right] \quad (1)$$

where $good(n)$ is 1 only if node n is fault free.

Theorem 2 describes reconfigurability at the SST level. Corollaries 1-3 provide necessary and sufficient conditions for reconfiguration of a failure anywhere in a SOFT tree. Theorem 3 describes the class of fault subsets which are reconfigurable in SOFT architectures.

THEOREM 2: The failure of any node n within an SST whose root is not displaced is reconfigurable if and only if $SS(n)$ is 1.

PROOF: From Definition 3, for a node in SST S to have SS of 1, either (A) its associated spare is available, or (B) the associated spare of S is faulty and its nonassociated SST has SS of 1, or (C) there exists an SST whose spare is available such that all SSTs 'left' of S and 'right' of the available spare have either faulty spares or spares configured in to their associated SSTs. If (A) is true, then the failure is reconfigurable (see Lemma 4). If (A) is not true but (B) is true, then either reconfiguration is possible through the failed spare into its nonassociated SST, by the above analysis and Definition 3, or (C) is true. If (C) is true (but (A) and (B) are not true), then the nonassociated spare of S must be configured into S. If the SS of S is 1 then at most one fault has occurred in S prior to this failure, by Definition 3. Lemma 4 indicates that configuration of the nonassociated spare is possible. If the nonassociated spare was not being used, then the reconfiguration is finished. Otherwise, the nonassociated spare had been configured in as a son of its associated SST. If the nonassociated spare has failed then reconfiguration proceeds by bypassing the

spare, as though the SST were larger (using the switches of Figure 2c). Since *SS* of this spare was 1, reconfiguration can proceed toward its nonassociated spare, by the same analysis. This continues until the available spare is configured into the array. The algorithms of Section C guarantee that a nonassociated spare is configured in only if there is no configuration such that only associated spares are configured in. As a result, if a spare is configured in to its nonassociated SST, then further reconfiguration cannot proceed in this direction without double displacement, thus satisfying the necessary condition. □

COROLLARY 1: Failure of node n, above the SST level, is reconfigurable if and only if the node is undisplaced and $SS(n)$ is 1, or, the node is displaced and $s(n)$ is 1.

PROOF: The proof considers the displaced and undisplaced conditions separately. If an undisplaced node n above the SST level fails, then it reconfigures toward the root of an SST. If $SS(n)$ is 1 then there is a path through fault free nodes to an SST which has *SS* of 1, which follows from Definition 3. Since failure of the root of an SST and its displacement are equivalent below the SST level, this is reconfigurable by Theorem 2. If $SS(n)$ is 0 then either there is no path through fault free nodes to the SST level, in which case the failure is not reconfigurable by Lemmas 1 and 2, or such a path exists only to the root of an SST with SS of 0 which is not reconfigurable by Theorem 2. If n is displaced, the failure of n is tolerable if the displacement can be moved into another sub-tree. When n fails, if b_n can assume f_n's position, then n's displacement can be shifted into the b_n sub-tree. By Lemma 3, b_n can not already be displaced. As above, b_n can assume its father's logical position if and only if $SS(b_n)$ is 1. Shifting the displacement to b_n is the only alternative if f_n is the failure creating n's displacement. If f_n has not failed, then an alternative is to shift the displacement into b_{f_n}. This can occur if and only if $SS(b_{f_n})$ is 1. Similarly, if neither f_n nor f_{f_n} has failed, the displacement can be shifted to the brother of f_{f_n}. Consequently, the displacement can be shifted to the brother of any ancestor, A, if and only if b_A has *SS* of 1 (as above) *and* all ancestors of n which are on level below A are fault free. □

COROLLARY 2: Failure of a non-redundant node n within SST S can be tolerated if and only if either $SS(n)$ is 1, or the root of S, r, is displaced and $s(r)$ is 1.

PROOF: When a failure in an SST is detected, there are three possibilities for reconfiguration. If and only if $SS(n)$ is 1, reconfiguration can proceed toward either the associated or the nonassociated spare, from Theorem 2. If r has been displaced, then the third alternative is to shift r's displacement into another SST, and use S's associated spare to reconfigure for n's failure. From Corollary 1, this is possible if and only if $s(r)$ is 1. □

COROLLARY 3: The failure of a spare n is tolerable if and only if the failure of a non-redundant node in either the associated or unassociated SST is tolerable.

PROOF: If the spare is not configured into the array, then its failure is tolerable. A failure in either SST adjacent to the SST is also tolerable. If the spare has been configured into the array, then there are two possibilities for reconfiguration. The first is to undo the displacement causing the spare to be configured into the array. The second is to bypass the spare and use the neighboring SSTs other adjacent spare. To do the first, it is necessary and sufficient that the spare had been configured into its associated SST (if the spare is configured into its unassociated SST, then the displacement cannot be shifted out of that SST) and a failure within the SST is tolerable (this follows from Lemma 4 and Corollary 2). If the spare is to be bypassed, then it is necessary and sufficient that a failure in the neighboring SST be tolerable, also by Lemma 4 and Corollary 2. □

THEOREM 3: A SOFT tree is properly reconfigured if and only if each faulty node above the SST level has a path through fault-free nodes to the ith level (i.e., no subsets such as in Figure 5), and there are no more than $x+1$ faults or displacements of the root of an SST in any x adjacent SSTs, and there are at least $x+1$ spares in the tree.

PROOF: If there is no failure such that there is no path through fault-free nodes to the roots of the SSTs, then reconfiguration above the SSTs can proceed through fault free nodes into the roots of the SSTs (from Corollary 1). The question of reconfigurability then concerns only the availability of spares at the SST level. If there are $x+1$ faults in x SSTs, there are exactly $x+1$

spares available. At most two failures or a failure and a displacement of the root can occur within an SST, otherwise the condition of the Corollary is violated for $x=1$. Lemma 4 indicates that any two failures within an SST can be tolerated as long as the nonassociated adjacent spare can be reconfigured in. Since there are $x+1$ spares available, the faulty PEs are reconfigurable. □

B. Tolerance of Switch and Link Failures

In this section, a SOFT architecture is shown to be capable of tolerating functional failures in communication links and stuck-open and closed faults in the switches of Figure 2. The following abbreviations are used in describing each node: PFF denotes the processor of a node as being fault free, SFF represents the switches associated with a node as being fault free, and LFF indicates the links connected to a node are fault free.

1) Failure of Links

DEFINITION 4: A node of a tree is *replaceable* if and only if it is PFF, and, in the given fault scenario, the failure of that processor is reconfigurable. □

The difference between replaceability and *SS* is that replaceability assumes that there are no link/switch failures to prevent reconfiguration. Similarly, replaceability is substituted for spare sufficiency in the definition of shiftability, $s(n)$.

DEFINITION 5: A *nonredundant link* l (link between n and f_n) is *isolated* if: A) f_l, *left* son_{f_l}, and *right* son_{f_l} are SFF and LFF (except for the failed link), and B) b_{son_l} is replaceable. □

LEMMA 5: If a faulty nonredundant link is isolated, then the tree can be reconfigured around that link.

PROOF: The switching scheme of Figure 2 allows the faulty link to be removed from the tree if b_{son_l} can be displaced. The restriction of Definition 5 (B) guarantees that b_{son_l} can be displaced. Part A of Definition 5 guarantees that the switching around the failed link will allow this

reconfiguration to occur. □

If no ancestor has failed, then the link failure is considered as a failure of f_l. If an ancestor has failed, then its displacement is shifted through f_l thereby freeing a spare in another SST for use in case of another failure.

DEFINITION 6: A redundant link l, in an upper level, is isolated if: A) f_{b_l} is PFF, SFF, and LFF, and both brothers are SFF, and B) $s(f_{b_l})$ is 1. □

LEMMA 6: If a redundant link, at upper levels, is isolated then its failure is reconfigurable.

PROOF: The restrictions of the definition, and Corollary 2, guarantee that the displacement which necessitates the use of the redundant link can be shifted into another subtree. Thus, the faulty redundant link is no longer configured into the tree. □

DEFINITION 7: A redundant link l in level i is isolated if either: A) the displacement is to the left (right) and the left (right) brother is replaceable, or B) the displacement is from above and f_{b_l} is PFF, SFF and LFF, and $s(f_{b_l})$ is 1. □

Consequently, if f_{b_l} has failed then the displacement is not shiftable.

LEMMA 7: If any redundant link is isolated then its failure is reconfigurable.

PROOF: At upper levels this is true by Lemma 6. At level i, Definition 7 applies. If the displacement is to the left or right (Definition 7A), and if that leaf is replaceable, the displacement which caused the use of this redundant link can be reversed to configure in a different spare. Thus, the redundant link once again becomes isolated from the tree. If the reconfiguration involving the redundant link is a result of a displacement of the father, then the displacement of the father must be shifted into another subtree, *and* reconfiguration in level i must be in the direction away from the faulty link, otherwise the faulty link will not be configured out of the tree (B). This is possible under the condition of part B of Definition 7 by Corollary 2. □

THEOREM 4: Isolated link failures can be tolerated in SOFT architectures.

PROOF: From Lemmas 5 and 7, all inter-processor isolated link failures can be tolerated. A fault in a link connected directly to a processor (i.e., links A and B in Figure 2a) can be modeled as a failure of the processor itself. In this case, the link fault is reconfigurable if the processor is replaceable. Failure of link C in Figure 2a only prevents displacement to the left if the PE has failed (displacement to the right is possible), or displacement to the right if the PE has not failed (displacement to the left is still possible). In Figure 2b, failure of link C prevents displacement of the node to the left or right if the node is not faulty, and to the left only if node has failed. □

2) Failure of Switches

DEFINITION 8: Any *switch in a nonspare node* is defined to be *isolated* if it is in a node which is PFF, SFF (except for the faulty switch), LFF, and the node is replaceable. Any *switch in a spare node* is defined to be *isolated* if the spare is SFF (except for the faulty switch), LFF, and the replaceability of the SSTs adjacent to the spare are 1. □

THEOREM 5: If a switch is isolated, then its failure is reconfigurable.

PROOF: By symmetry, the only switches which need to be considered are switches 1 - 5 in Figures 2a and 2b and the switches of 2c. Since the node is PFF, SFF, and LFF prior to the switch failure, the node can support any reconfiguration for which either the faulty switch is not necessary (stuck-open fault), or the switch should be closed, or the closing of the switch has no effect. The fact that the node is replaceable indicates that the failure of this node is tolerable. As a result, if the node is displaced it can be returned to the undisplaced state. This can be done at upper levels by shifting the displacement into another subtree. At the leaves, the reconfiguration is shiftable since the failure of this leaf is reconfigurable. Failure of the switches of Figures 2a and 2b are tolerable by the following analysis:

(1) *Switch 1:* A stuck-open fault is reconfigurable as a failure of the processor. Reconfiguration of stuck-closed is not required since the node is PFF.

(2) *Switches 2,3,4,5:* Stuck-open fault is reconfigurable by placing the node in the undisplaced state. Stuck-closed can also be properly configured in the undisplaced state.

In spare switching (Figure 2c), for the case of stuck-closed faults, the spare is not configured in unless it is needed. Consequently, stuck-closed faults have no effect. For stuck-open faults, if the spare is not configured in, then the replacebility in both SSTs is certainly 1, and failure of any switch is tolerable. If the spare is configured in, then a stuck-open fault is tolerable since either SST can reconfigure a different spare in, which means that there is a configuration which does not employ these switches. □

C. Reconfiguration Algorithms

In the first section, the reconfiguration algorithms are presented assuming that the links and reconfiguration mechanism are fault-free. In Section 2, the algorithms presented in this section are extended to include link and switch fault tolerance. Finally, failure of the control mechanism is considered in Section 3.

1) Reconfiguration of Node Failures.

The reconfiguration algorithm for high-level nodes is presented first. This corresponds to all processing elements above the SST level. If the SOFT tree has been allocated one spare per leaf, or ≈50% spares, then all nodes except for the leaves and spares follow this algorithm. Reconfiguration algorithms for SST nodes, leaves, and spares are presented next. The algorithms are appropriate for SSTs of arbitrary size.

The five basic configurations for upper level nodes are shown in Figure 6. Figure 6a shows the normal or starting configuration of all nodes. Figure 6b depicts a node which has been displaced. In this section, the configuration for a displaced node is the same, independent of which son is displaced. Determination of which son is displaced (and the subsequent switch settings) is done within the algorithms. Figure 6c shows a node whose brother has been displaced. By Lemma 3,

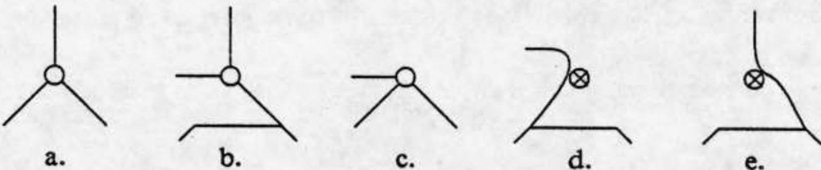


Figure 6. Switch configurations for upper level nodes.

only one of two brothers can be displaced at any one point in time. Figures 6d and 6e represent failed nodes whose brother is and is not displaced, respectively.

A simple technique for controlling the configuration of the switches is to associate three state variables with each node: *good*, *disp*, and *fatherdisp*. The variables are defined as follows:

good : True if and only if the node is functioning properly.

disp : True if and only if the node is displaced (see Definition 2).

fatherdisp : True if and only if the node's brother has been displaced.

The node's configuration is based on Boolean expressions involving these three variables. Table 2 gives the configurations from Figure 6 and the conditions under which each configuration occurs.

The control of the internal state variables requires some form of communication between nodes. For this purpose, three signals are defined: *recon*, *brorecon*, and *unrecon*.

recon : *Recon* is issued from a displaced or failed father to the son which is to assume the identity of the father in the reconfigured array.

Table 2. Configuration versus Expression for High-level Nodes.

Figure	Expression
6a.	$good \cdot \overline{disp} \cdot \overline{fatherdisp}$
6b.	$good \cdot disp$
6c.	$good \cdot fatherdisp$
6d.	$\overline{good} \cdot fatherdisp$
6e.	$\overline{good} \cdot \overline{fatherdisp}$

unrecon : *Unrecon* is issued from a son to its father and brother to indicate that the father should no longer be displaced.

brorecon : *Brorecon* is issued from the brother or father of a node to indicate that its brother has been displaced.

Additionally, a signal (*fail*) which is internal to the node is required which indicates that the failure of the node has just been detected.

Two global variables are used to govern the direction of reconfiguration: spare sufficiency (*SS*) and *path*. *SS* in these algorithms is defined below, and is not identical to the *SS* defined in Section A, although the intuitive meanings are similar. *SS* and *path* are recursively defined as follows.

SS: *SS* of a leaf is 1 if and only if the processor is fault-free *and* its associated spare is both fault-free and not configured into the array. *SS* of nodes in levels 0 to $i-1$ are 1 if and only if *SS* of the node's left or right son is 1 *and* the node is fault-free.

path: The *path* of a leaf is one if the leaf is fault-free. The *path* of all other nodes is one if and only if the node is fault-free and the *path* of its left or right son is one.

With these parameters, a reconfiguration algorithm for upper level nodes has been derived, and is presented as Algorithm 4. A subscript such as SS_{right} denotes the spare sufficiency of the right son of a given node. The function of the algorithm is to wait for a signal and then respond (reconfigure) appropriately. Each of the five if statements corresponds to signals. For instance, if a node receives a *recon* signal, then it sets its own switches and issues a *recon* to one of its sons on the basis of its internal and global variables.

For Algorithms 4 and 5, the definitions of *path* and *SS* assign the *leaves* values on the basis of their internal variables, and the upper level nodes formulate their values for these variables by ORing their sons' values. If SSTs have more than one node, then the root of the SST takes the place of leaves in those definitions. Thus, only nodes in levels above the SSTs follow Algorithm 4.

ALGORITHM 4: 50% SPARING.

While true do

  if *recon*
    then if $\overline{good}$ then *tree fails*
      set *disp* ; issue *brorecon*
      if $SS_{left}+\overline{SS_{right}}\cdot path_{left}$
        then issue *recon* to left son
        else issue *recon* to right son

  if *brorecon*
    then set *fatherdisp*

  if *unrecon*
    then if *signal is from brother*
        then clear *fatherdisp*
        else clear *disp*
          if *good*
            then issue *unrecon*
            else set *fail*

  if $fail \cdot disp$
    then clear *good* ; clear *disp* ; issue *unrecon*

  if $fail \cdot \overline{disp}$
    then clear *good*
      if $SS_{left}+\overline{SS_{right}}\cdot path_{left}$
        then issue *recon* to left son
        else issue *recon* to right son

  end.

Reconfiguration of nonleaf nodes within SSTs is similar to the reconfiguration just described. In Algorithm 4, reconfiguration proceeds in the direction of the left son where possible. For SST nodes, the opposite is the case. The *path* variable is redefined, and spare availability (*SA*) takes the place of *SS*.

path: *path* for a leaf node is defined as one if and only if the leaf is fault-free *and* it is not displaced (to the left, right, or up). *path* of all other SST nodes is high if the *path* of its left or right son is high and the node is fault-free.

SA: *SA* is high if and only if the SST's associated spare is fault-free *and* not in use.

rootdisp: *rootdisp* is true if and only if the root of the SST has been displaced.

While both *SA* and *rootdisp* depend only on the status of one node, this does not imply that separate lines need to be run between the spare or the root and all nodes within the SST. Alternatively, the signals could be passed between nodes in the same manner as *SS* and *path*. The

reconfiguration procedure for nonleaf SST nodes is presented in Algorithm 5.

The algorithm for leaf nodes is presented for the general case. The algorithm is appropriate for arbitrarily large SSTs. Simplifications exist for small SSTs (25 - 50% sparing). With 25-50% sparing, the leaf algorithm becomes similar to the SST algorithm.

The possible line configurations for leaf nodes are shown in Figure 7. Since the possible switch configurations are unique from those at upper levels, a new set of internal state variables is defined.

ALGORITHM 5: SST NODES.

While true do
    if *recon*
        then if $\overline{good}$ then *tree fails*
            set *disp* ; issue *brorecon*
            if $SA + \overline{path_{left}} + rootdisp$
                then issue *recon* to right son
                else issue *recon* to left son
    if *brorecon*
        then set *fatherdisp*
    if *unrecon*
        then if *signal is from brother*
                then clear *fatherdisp*
                else clear *disp*
                    if *good*
                        then issue *unrecon*
                        else set *fail*
    if $fail \cdot disp$
        then clear *good* ; clear *disp* ; issue *unrecon*
    if $fail \cdot \overline{disp}$
        then clear *good*
            if $SA + \overline{path_{left}} + rootdisp$
                then issue *recon* to right son
                else issue *recon* to left son
    end.

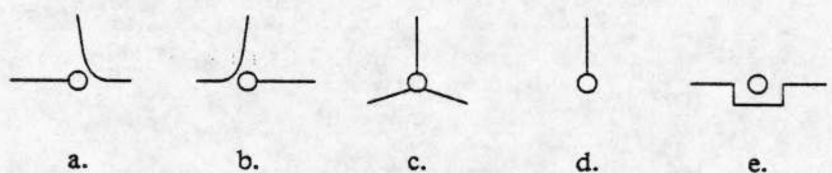


Figure 7. Switch configurations for leaf nodes.

$disp_{right}$: $disp_{right}$ indicates that the leaf is in the displaced state to the right, i. e., it is assuming the identity of its right neighbor.

$disp_{left}$: as above to the left.

When $disp_{right}$ and $disp_{left}$ are both high, this indicates that the node is displaced to level $i-1$.

good : as before.

The leaf node's configuration is based on Boolean expressions of these three variables. Table 3 enumerates the configurations of Figure 7 and the conditions for their use.

Similarly, the internode signals need to be redefined.

$recon_{dir}$: $recon_{dir}$ is as recon, in the direction (left, right, or up) indicated by dir, i. e., $recon_{up}$ is a request, from father to leaf, to displace the node in order to replace its father.

unrecon : *unrecon* has the same function; however, it is issued only by leaves which are right sons and is sent to both the node's father and brother.

brorecon : as before.

fail : as before.

Using these parameters, Algorithm 6, on page 28, was derived. As in the previous algorithms, signals corresponding to the if statements are input to the node, and the node's configuration and output signals react on the basis of these.

Table 3. Configuration versus Expression for Leaf Nodes.

Figure	Expression
7a.	$disp_{right} \cdot \overline{disp_{left}}$
7b.	$disp_{left} \cdot \overline{disp_{right}}$
7c.	$disp_{left} \cdot disp_{right}$
7d.	$good \cdot \overline{disp_{right}} \cdot \overline{disp_{left}}$
7e.	$\overline{good} \cdot \overline{disp_{right}} \cdot \overline{disp_{left}}$

Figure 8 depicts the possible spare configurations, and Table 4 indicates the expressions which must be true for each configuration to be realized. Algorithm 7 presents the control strategy for spare reconfiguration (See p. 28.).

The reconfiguration algorithms require communication between adjacent nodes. This includes both updating values such as *SS* and *path* as well as issuing signals such as *recon* and *unrecon*. While separate lines could be allocated for these signals, it is also possible to allocate the data lines for these purposes.

A similar issue is whether to allocate separate hardware for reconfiguration or to implement the algorithms in soft/firmware. Implementing the algorithms in hardware is straightforward. To implement the algorithms (and diagnosis) in software, however, requires that a failed node never perform diagnosis or reconfiguration. This can be accomplished by having the father perform diagnosis and reconfiguration for both of its sons. Thus, the sons would have registers for their switch configurations and no other hardware to determine its switch configuration. When a node failure on level i has been discovered, the failing node's father, on level $i-1$, reconfigures the node's switches and then performs reconfiguration for its new son on level $i+1$. This requires mild modification of the preceding algorithms, but is straightforward.

ALGORITHM 6: LEAF NODES.

While true do
if $recon_{right}$
then if $(not\ adjacent\ to\ spare) + \overline{disp_{left}}$
then set $disp_{right}$; issue $recon_{right}$
else issue $unrecon$; clear $disp_{left}$
if $brorecon$
then if $node\ is\ a\ left\ son$
then if $disp_{right}$ then issue $recon_{left}$
if $good$
then set $disp_{left}$; clear $disp_{right}$
else clear $disp_{right}$; clear $disp_{left}$
else set $disp_{right}$
if $\overline{good}$ then clear $disp_{left}$; clear $disp_{right}$
if $recon_{up}$
then if $good \cdot \overline{disp_{left}}$
then if $node\ is\ a\ right\ son$
then if $\overline{disp_{right}}$ then issue $recon_{right}$
else issue $recon_{left}$
set $disp_{right}$; set $disp_{left}$; issue $brorecon$
else *tree fails*
if $recon_{left}$
then if $disp_{right} \cdot \overline{disp_{left}}$
then clear $disp_{right}$; issue $recon_{left}$
if $good$
then clear $disp_{left}$
else set $disp_{left}$
if $\overline{disp_{right}} \cdot disp_{left} + \overline{disp_{left}} \cdot \overline{disp_{right}} \cdot \overline{good}$
then *tree fails*
if $disp_{left} \cdot disp_{right}$
if $node\ is\ a\ right\ son$
then clear $disp_{right}$; clear $disp_{left}$
issue $unrecon$
else *tree fails*
if $\overline{disp_{right}} \cdot \overline{disp_{left}} \cdot good$
then set $disp_{left}$; issue $recon_{left}$
if $unrecon$ (implies node is a left son)
then if $good$
then clear $disp_{left}$
else set $disp_{left}$
if $fail \cdot disp_{right} \cdot \overline{disp_{left}}$
then issue $recon_{left}$; clear $good$
if $fail \cdot disp_{left} \cdot \overline{disp_{right}}$ then *tree fails*
if $fail \cdot disp_{left} \cdot disp_{right}$
if $node\ is\ a\ right\ son$
then issue unrecon; clear $disp_{left}$; clear $good$
else *tree fails*
if $fail \cdot \overline{disp_{left}} \cdot \overline{disp_{right}}$
then if $SA + rootdisp$
then issue $recon_{right}$; set $recon_{right}$; clear $good$
else issue $recon_{left}$; set $recon_{left}$; clear $good$
end.

Table 4. Configuration versus Expression for Spare Nodes.

Figure	Expression
8a.	$good \cdot \overline{disp_{right}} \cdot \overline{disp_{left}}$
8b.	$good \cdot disp_{right}$
8c.	$good \cdot disp_{left}$
8d.	$\overline{good}$

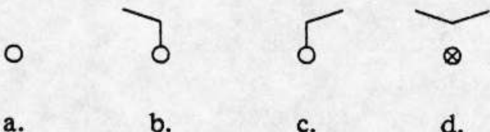


Figure 8. Switch configurations for spare processors.

ALGORITHM 7: SPARES.

While true do
if $recon_{right}$
then if $good$
then set $disp_{right}$
else issue $recon_{right}$
if $recon_{left}$
then set $disp_{left}$
if $disp_{right} + \overline{good}$
then issue $recon_{left}$; clear $disp_{right}$
if $fail \cdot disp_{left}$
then issue $recon_{left}$; clear $good$
if $fail \cdot disp_{right}$
then clear $good$
if $SA_{right} + rootdisp_{right}$
then issue $recon_{right}$
else issue $recon_{left}$; clear $disp_{right}$
end.

---

2) Reconfiguration for Link and Switch Failures

As with processing elements, the fault model employed for links and switches is a functional fault model. Stuck-at faults and shorting of lines have been considered. The fault model for switches consists of stuck-open and stuck-closed. Such a fault model applies to faulty input at the switch control as well as to failure of the switch.

In Figures 6b, 6d, and 6e, a displaced or bypassed node is shown as configuring the switches such that the son or bypass data are passed on to both sons. Naturally, this is unacceptable if link

fault tolerance is to be provided (although it allows the three internal state variables to be set mnemonically). Instead, the switches must be set based on the direction of reconfiguration. Thus, the number of possible switch configurations grows from five to eight. Three internal state variables are, therefore, still sufficient. A switching scheme which allows for tolerance of link failures was presented in Figure 2. The corresponding modifications to Algorithms 4 and 5 consist only of setting the internal state variables to distinguish the direction of reconfiguration.

Reconfiguration around a failed link l can be implemented by:

1. Clear *path* and *SS* of the son of the link.
2. Set *fail* of father of the link.

If an ancestor of f_l has failed, then the corresponding displacement can be shifted through f_l, thereby avoiding the use of an extra spare. Assuming link failures occur with relatively low probability, however, this may not be worth the overhead. Although both the brother and father links of a node cannot fail without the tree failing, it should be noted that even if these two lines are shorted together the tree can still function. In this case, setting *fail* of the son/brother of the shorted links results in correct reconfiguration.

Failure of a redundant link implies that it is either in use or attempted use. Reconfiguration for failure of a redundant link can be implemented as follows:

1. Clear *path* and *SS* of both nodes adjacent to the link.
2. Set *fail* of the displaced node.

If the spare which is configured in to replace the node is needed, then an *unrecon* signal will travel from the leaves up to the node whose brother link has failed. At this point, the node can be reconfigured back into the array.

Failure of the lines within the nodes must also be considered. Failure of lines such as A and B of Figure 2a can clearly be modeled as failure of the PE itself. Failure of C is handled in the

following manner:

1. Clear $path_{right}$.

2. Set *fail* of the PE.

These modifications apply to the switching scheme of upper level nodes. The modifications for leaves and spares are similar but with less complexity due to their simpler switching structure.

In the stuck model of switch failure, the simplest way to deal with a stuck-open fault of a switch is to handle it as a break in the internode line. For the case of stuck-closed faults, the only time a stuck-closed fault requires an alternate reconfiguration strategy is when the switch is closed but the current switch configuration requires the switch to be open. Taking advantage of symmetry, switches 1 through 5 of Figure 2a will be considered. If switch 1 is stuck-closed, this is only an issue if the PE has failed. This implies that the line to the left son is no longer reliable. Thus, the steps to tolerate this failure are the same as the link failure. If switch 2 is stuck-closed, then it, too, can be modeled as a failure of the line to the left son. Significantly, this is only an issue if displacement to the left or brorecon is attempted. Similarly, switch 3 being stuck-closed can be handled by considering the PE to have failed displacing through the right son (set $path_{left}$ and SS_{left} to zero). Switch 4 being stuck-closed can be handled by treating the PE as faulty. Similarly, switch 5 can be handled by considering the PE to have failed. Similar results have been obtained for leaf switching.

3) Reconfiguration for Control Failures

The final area of switch and link failure to be considered concerns failure in the reconfiguration control hardware. If a soft/firmware approach is used, this is not as much of an issue, since only good processors perform the reconfiguration algorithms. The assumption that switch and link failure is isolated in its incidence is extended to control failure as well. For instance, if a node's *good* variable is stuck-at-one, then there is little that can be done if the node fails. This corresponds to an undetected fault, which cannot be tolerated.

The fault model employed for control failure is also a functional fault model. It considers the case where a value or signal is one or zero when it should not be. The technique for handling *good* stuck-at-zero is self-evident.

If *disp* is stuck-at-one then the response should be to clear *good*. If *disp* is stuck-at-zero, then its *path* and *SS* should be set to zero. If fatherdisp is stuck high, then this can be modeled as failure of the father. If it is stuck-at-zero, then the path and SS of its brother should be set to zero.

If a global variable is zero when it should be one, then reconfiguration will not proceed in that direction. This is not significant, since the worst situation is the node failing, in which case reconfiguration cannot proceed in that direction anyway. If a variable is stuck-at-one then, under the assumption that switch and link failure are isolated in occurrence, the second variable is valid, i.e., if *SS* is stuck-at-one then, if reconfiguration cannot proceed in this direction, path will still be zero. Consequently, both *SS* and *path* must be checked before selecting a son for displacement.

Finally, reconfiguration for signal failures (e.g., *recon*, *brorecon*, ...) are considered. Since each (upper level) node has two *recon* signals, one to each son, if one *recon* is stuck low, then reconfiguration simply cannot proceed in that direction (i.e., *SS* and *path* are zero). If a *recon* is stuck high, then the node should be set as faulty. Thus, the recon signal can be ignored after that. Similarly, if *unrecon* is stuck high, then the node can be set faulty, further *unrecon* signals then being invalid. If *unrecon* is stuck low, then the node can be set faulty, thereby assuring that the *unrecon* need never be used. If *brorecon* is stuck low, then path and SS of the node must be set to zero. If it is stuck high, then this can be modeled as a failure of the father, with reconfiguration proceeding in the direction of the node.

D. Analysis for Static Reconfiguration

The algorithms presented in Section C are appropriate for dynamic reconfiguration of failures. Static reconfiguration, where a tree has a fixed subset of nodes which are faulty, requires reconfiguration and analysis of reconfigurability for the entire structure given a subset of faulty

nodes. An algorithm has been developed which determines if an input fault subset is reconfigurable. The algorithm has been implemented in Pascal.

Input to the algorithm is as follows. First, the faulty nodes in levels above and including the root of the SSTs are specified. Next, the associated spare of each SST is specified as being faulty or fault-free. Finally, the number of nonspare faults in each SST is input to the algorithm.

Nine Boolean variables are assocated with each node at and above the level of the roots of the SSTs. The algorithm begins by determining the reconfigurability of the tree at the SST level alone. This is done by initializing the Boolean values for the root of each SST, on the basis of the status of the SST's root, and the number of faults within the SST. These values are then altered based on the Boolean values of the node's neighbors (brothers and cousins) on the same level. If the tree has not been determined to be unreconfigurable, then the nodes on the level above the SST level are considered. The Boolean values of any node above the SST level are set on the basis of its son's Boolean values. These values are then altered on the basis of the Boolean values of the other nodes on the same level.

Since the reconfigurability of the tree can be determined at each level, based only on the Boolean values from the level below, and the status of nodes on that level, the algorithm runs in time directly proportional to the number of nodes above the Spare SubTree level. Since the size of the tree above the SSTs equals the number of spares less one, the algorithm is $O(n)$ for a fixed percentage of spares, where n is the number of nodes in the tree. For a fixed number of spares, the algorithm runs in order constant time. If the tree has been determined to be reconfigurable, then a proper reconfiguration for the tree can be determined on the basis of the Boolean values associated with each node.

IV. RELIABILITY ANALYSIS

In this section it is shown that the reliability of a SOFT binary tree, even with the restrictions imposed by VLSI layout, is always superior to a tree implemented using the class of approaches employed by [7, 8]. This is demonstrated by first establishing an upper bound on the reliability of the previous approaches, independent of the actual implementation, i.e., their optimal reliability, and comparing it to a lower bound for SOFT trees. Exact reliability calculations of some specific SOFT implementations are also derived and compared.

A. Reliability of Other Approaches

It is assumed that the $i+1$ level tree is allocated k spares. A *Modular Sparing Approach* (MSA) to fault tolerance in binary trees is any approach to reconfigurable design which partitions a tree into k groups of processors and allocates each group of processors one spare to be used exclusively by that group. The work of Raghavendra, et al. (RAE) and Hassan and Agarwal (M-trees) can both be classified as MSA. Significantly, SOFT trees are not included in this category. It should be noted that the strategy of Rosenberg [9] is not MSA. However, Rosenberg's strategy does not allow for interconnect or switch failure. With an MSA, each module must be functioning in order for the tree to be functioning. Thus, the reliability can be expressed as the product of the reliability of all of the modules. In general this is:

$$R_{sys} = \prod_{m=1}^{m=k} R_m \qquad (2)$$

where R_m is the reliability of the mth module. Although some MSA schemes may tolerate interconnect failure, the following reliability analysis considers only processor failures, for sake of simplicity. Since the spare can be configured into the module in case of any single failure, the reliability of each module can be expressed as:

$$R_{module} = R^q + qR^{q-1}(1-R)$$

where R is the reliability of each individual processor and q is the number of processors in the module, including the spare. The reliability for all processors is assumed to be equal and exponentially distributed (i.e., $R=e^{-\lambda t}$). Although this assumption is not accurate for many environments, it does provide an initial point of comparison and is a common assumption in reliability analysis [7, 8, 16]. For simplicity, the failure rate of spares, μ, is assumed to be equal to the failure rate of nonredundant processors, λ. In the following discussion, it is assumed that the trees can be divided evenly into modules of size q, although the theorem does not rely on this assumption.

THEOREM 6: Optimal reliability in an MSA tree corresponds to when the tree is divided into modules of equal size: $q=(2^{i+1}-1+k)/k$.

PROOF: Consider a tree with equal module size. The following inequality indicates that every time a single node is moved from a module of size q to another module of size q, thereby creating modules of size $q+1$ and $q-1$, R_{sys} decreases:

$$[R^q+qR^{q-1}(1-R)]^2 > [R^{q+1}+(q+1)R^q(1-R)][R^{q-1}+(q-1)R^{q-2}(1-R)]$$

Additionally, the following inequality indicates that moving nodes from modules of smaller size into modules of larger size will decrease reliability:

$$[R^{q+c}+(q+c)R^{q+c-1}(1-R)][R^{q-k}+(q-k)R^{q-k-1}(1-R)] >$$
$$[R^{q+c+1}+(q+c+1)R^{q+c}(1-R)][R^{q-k-1}+(q-k-1)R^{q-k-2}(1-R)]$$

for any $0 \leqslant c$, and $0 \leqslant k < q$. □

It has been shown that the reliability versus spares curve of M-trees is greater than that of RAE [8]. The reason for this is that the RAE scheme allocates an entire spare to the root, whereas M-trees can 'spread' the spare out into level 1 (or more) nodes. At lower levels, however, the number of nonredundant nodes per spare are the same. Consequently, as i and/or k increases, the reliabilities of both schemes converge. In contrast, the next section demonstrates that the SOFT

approach *always* results in superior reliability over MSA designs.

B. Reliability of SOFT Trees

The reliability of a redundant system composed of nodes with equal reliability R, which is not subject to degraded performance, can be thought of as a polynomial of degree $(2^{i+1}-1+k)$ where there are $(2^{i+1}-1+k)$ nodes in the system. The polynomial can be expressed as:

$$R_{sys}=\alpha_0 R^{(2^{i+1}-1+k)}+\alpha_1 R^{(2^{i+1}-1+k)-1}(1-R)+\cdots+\alpha_j R^{(2^{i+1}-1+k)-j}(1-R)^j+\cdots$$
$$+\alpha_{(2^{i+1}-1+k)}(1-R)^{(2^{i+1}-1+k)}$$

where R is the reliability of individual components, and α_j is the number of ways in which j reconfigurable faults can occur in the tree ($\alpha_j=0$ for $j>k$). For comparison of α, an α_j from a SOFT reliability equation is denoted as α_{j_s}, whereas, α_{j_m} is associated with the optimal MSA reliability.

In order to analyze SOFT reliability, it is necessary to calculate the number of possible processor locations for the jth faulty processor to occur given that $j-1$ faults have already been successfully reconfigured. This is dependent on the specific SOFT implementation, however, using the following identity, a lower bound on the number of possible locations for the jth fault is derived in Theorem 7.

$$\text{Number of nodes per SST (including the spare)} = \frac{2^{i+1}}{k}$$

If a given SOFT or MSA tree contains $j-1$ specific faults, and the tree has not failed, then the location of the faulty processors (fault subset) is referred to as an α_{j-1_s} or α_{j-1_m} scenario, respectively. In SOFT, if $j-1$ faults have been reconfigured, this implies that the associated spares of $j-1$ SSTs are configured in (or failed), and $k-j+1$ SSTs have associated spares which are not used.

THEOREM 7: $(k-j+1)\left[\frac{2^{i+1}}{k}+1\right]$ is a lower bound on the number of reconfigurable faults from an α_{j-1_s} scenario.

PROOF: In each of the $k-j+1$ SSTs which has its associated spare unused, at least $\left[\frac{2^{i+1}}{k}\right]$ positions for the jth fault exist, which can be reconfigured. If in α_{j-1_s} the father of one of the $k-j+1$ SSTs has not yet failed, then the failure of the father can be tolerated and 1 can be added to $\frac{2^{i+1}}{k}$ as the number of locations for $k-j+1$ of the SSTs. If the father has failed, however, then it must have been reconfigured into a neighboring SST, which means that a fault in that neighboring SST can be tolerated and the f_{SST}'s reconfiguration shifted into the SST with the free spare. The lower bound, therefore, remains valid. If two of the $k-j+1$ SSTs are adjacent to the same father, then father has been counted twice in the lower bound. If the father, f, has not yet failed, however, then the failure of f_f is also tolerable, and by the same analysis, the lower bound remains valid. □

THEOREM 8: The reliability of a SOFT tree is always greater than the reliability of an equivalent MSA tree for k (the number of spares) > 1.

PROOF: If it can be shown that for at least one j, $\alpha_{j_s} > \alpha_{j_m}$, and $\alpha_{j_s} \geqslant \alpha_{j_m}$ for all other j, then $R_{sys_{SOFT}} > R_{sys_{MSA}}$ (assuming that the reliability of individual processors is the same for both architectures). Since $\alpha_{0_m} = \alpha_{0_s} = 1$ and $\alpha_{1_m} = \alpha_{1_s} = (2^{i+1} - 1 + k)$, R_{sys} with $k=0$ or $k=1$ is identical for both schemes. Consider, however, α_2. MSA can tolerate only one failure per module. As a result:

$$\alpha_{2_m} = \binom{(2^{i+1}-1+k)}{2} - k\binom{(2^{i+1}-1+k)/k}{2}$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

and:

$$\alpha_{2_s} \geq \left\lceil \frac{(2^{i+1} - 1 + k)}{2} \right\rceil - k \left\lceil \frac{2^{i+1}/k}{2} \right\rceil$$

Now consider α_j. Since $\alpha_{2_s} > \alpha_{2_m}$, it will be shown by contradiction that $\alpha_{j_s} > \alpha_{j_m}$ given that $\alpha_{j-1_s} > \alpha_{j-1_m}$. From the definition of α, it is evident that if $\alpha_{j_m} \geq \alpha_{j_s}$, at least one fault configuration corresponding to a successful reconfiguration of $j-1$ faults in MSA (denoted as a fault configuration in α_{j-1_m}) must have more possible locations for successful reconfiguration of the jth fault *than* the possible locations for the jth fault in an α_{j-1_s} configuration.

Derivation of the number of 'choices' for the jth fault to occur for each α_{j-1} in optimal MSA trees is straightforward and is less than the lower bound of Theorem 7.

$$\text{places for } j \text{ per } \alpha_{j-1_m} = (k-j+1) \left\lceil \frac{(2^{i+1} - 1 + k)}{k} \right\rceil$$

□

Due to the analytical complexity of a global reconfiguration strategy and the variety of possible SOFT implementations, a closed form expression for reliability has not been found for arbitrary size trees with arbitrary numbers of spares. However, the analysis presented in Section III is sufficient to determine the reliability of any specific SOFT tree. The following section presents some exact reliability calculations for example SOFT implementations.

C. Reliability Examples

The reliabilities of four level trees, implemented by M-tree, RAE, and SOFT as a function of the number of spares, for $R = e^{-.1t}$, t=.5, and t=1.0, are shown in Table 5. It should be noted that the SOFT reliability is superior even when the modular schemes are allocated more spares. The data for t=1.0 is graphically displayed in Figure 9. In Table 6 and Figure 10, the reliabilities versus time curves of a tree with no redundancy, duplicated four level trees, and four level trees with four spares employing an optimal MSA, a SOFT approach, and a tree with optimal reliability are presented. A scheme with optimal reliability guarantees reconfiguration for any number of faults

Table 5. Reliability for a four-level tree by number of spares.

t	number of spares	R_{RAE}	R_{M-tree}	$R_{opt-MSA}$	R_{SOFT}
0.50	8	0.9504	———	0.9512	———
	5	0.9281	0.9350	0.9350	———
	4	0.9033	———	0.9248	**0.9974**
	2	———	0.8499	0.8811	**0.9528**
	1	———	0.8179	0.8179	**0.8179**
1.00	8	0.8274	———	0.8298	———
	5	0.7655	0.7832	0.7832	———
	4	0.7073	———	0.7563	**0.9665**
	2	———	0.6194	0.6553	**0.7841**
	1	———	0.5416	0.5416	**0.5416**

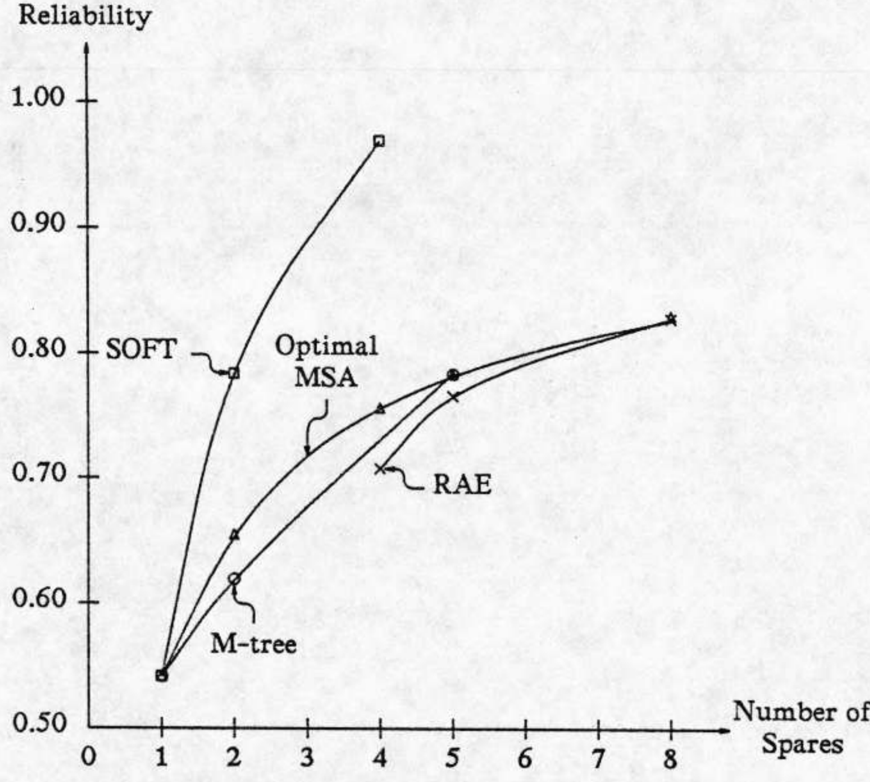


Figure 9. Reliabilities of four-level trees at t=1.0.

less than or equal to the number of spares. Table 6 includes calculations for an RAE tree with 4 spares and an M-tree with 5 spares. The M-tree approach was allocated five spares due to its inability to support four.

D. Increasing SOFT Reliability

There are several possibilities for enhancing the reliability of a SOFT architecture. If a designer is not concerned with VLSI layout issues or is willing to pay $O(n \log n)$ area, the SOFT tree can be implemented such that the leaves are fully connected and the full sharing of inter-SST spares is practical. As an alternative, sharing of spares between the $i=4$ leaf subtrees is possible using the procedure of Horowitz and Zorat [11]. A second option is the addition of redundant lines

Table 6. Reliability for λ=.1

t	$R_{no\ red.}$ 0 spares	R_{Dup}	R_{RAE} 4 spares	R_{M-tree} 5 spares	$R_{MSA-opt}$ 4 spares	$\mathbf{R_{SOFT}}$ **4 spares**	R_{OPT} 4 spares
0.00	1.000	1.000	1.000	1.000	1.000	**1.000**	1.000
0.25	0.687	0.902	0.972	0.982	0.979	**1.000**	1.000
0.50	0.472	0.721	0.903	0.935	0.925	**0.997**	0.998
0.75	0.325	0.544	0.810	0.866	0.847	**0.987**	0.990
1.00	0.223	0.396	0.707	0.783	0.756	**0.965**	0.971
1.25	0.153	0.283	0.604	0.694	0.661	**0.927**	0.936
1.50	0.105	0.200	0.505	0.604	0.566	**0.873**	0.886
1.75	0.072	0.140	0.417	0.517	0.447	**0.807**	0.822
2.00	0.050	0.097	0.339	0.437	0.396	**0.731**	0.747

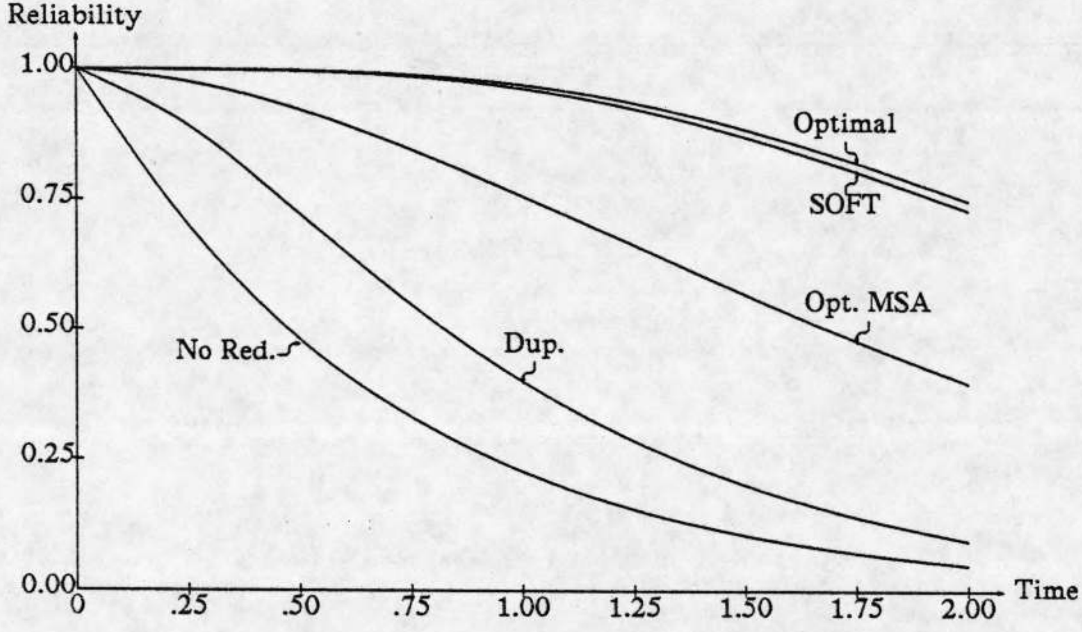


Figure 10. Reliability versus time for a four-level tree and four spares.

between leaves which are adjacent to the same spare. This allows reconfiguration of SSTs with numerous faults, assuming that the spares are available in neighboring SSTs. The cost of adding these lines is an additional k lines (where k is the number of spares). Finally, for applications such as yield enhancement, where processor yield may be quite low, ≈50% sparing is possible. In SOFT implementations with 50% sparing, spare and link placement is the same as in Algorithms 1 and 2, with the exception that each spare is associated with a single leaf and there are no cousin connections between leaves.

E. SOFT Performance Degradation

The SOFT approach, as with the RAE technique, allows for graceful degradation in performance. No redundant spares are allocated for graceful degradation, however redundant lines are located as described by Algorithm 2. For SOFT architectures a spare at the root is unnecessary, in contrast to previous graceful degradation approaches [7]. Since failures are passed to the leaves, the only nodes which must assume the functions of their brothers are the leaves. Also, as noted before, the redundancy in terms of links is reduced and the reliability is enhanced. The only failures which disable the tree are long runs of failures along the leaves and a failure of a node for which no path of good nodes into the leaves exist, i.e., the class of failures depicted in Figure 5.

V. SOFT N-ARY TREES

N-ary trees, in which each nonleaf node has N sons, are more suitable for certain tasks than classical binary trees. For example, 4-ary (quad) tree architectures have been proposed for implementing several classes of artificial intelligence related algorithms [17]. The following brief discussion summarizes how the SOFT approach is applicable to N-ary trees.

A. Construction of Reconfigurable N-ary Trees

1) Location of Redundant Lines

A link allocation approach which is applicable to arbitrarily large N is to restrict reconfiguration to displacement of the 'outside' two children of each non-leaf node. Redundant links to each of a node's brothers are added only to the two outside brothers. The link redundancy at upper levels is approximately $\frac{2N-3}{N}$%, which is O(1) instead of O(N).

2) Location of Redundant Processors

Allocation of up to one associated spare per group of N brothers is allowed. The spares are placed between SSTs with connections from a spare to both its associated SST and a neighboring SST. Redundancy is therefore:

$$\% \text{ link redundancy } \approx \frac{2N-3}{N} + \left(\frac{N^{i-1}+N^{c}}{\sum_{g=0}^{g=i} N^{g}} \right)$$

$$\% \text{ node redundancy } = \frac{N^{c}}{\sum_{g=0}^{g=i} N^{g}}$$

where N^c is the number of spares allocated to the tree, with $1 \leqslant c \leqslant i-1$.

B. Reconfiguration

Reconfiguration is fundamentally the same as in binary trees. A sample reconfiguration for four failures in a 5-ary tree is illustrated in Figure 11.

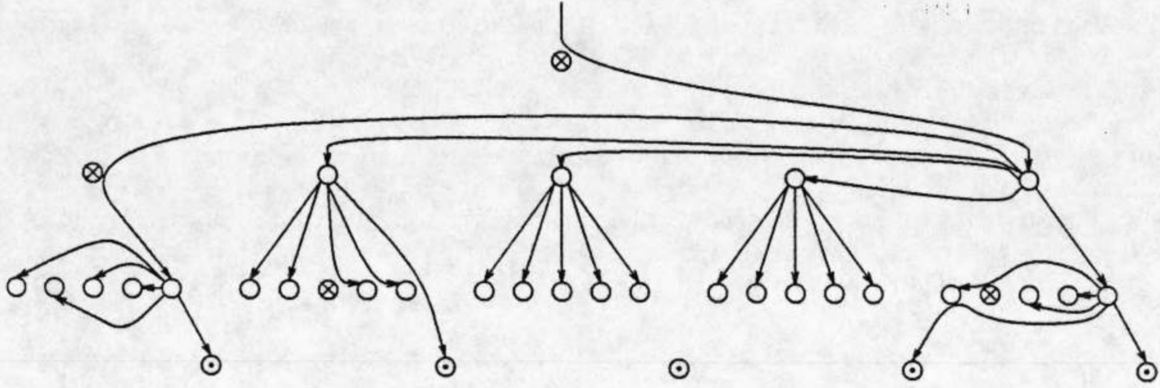


Figure 11. Reconfiguration in a 5-ary tree.

VI. CONCLUSIONS

A unique approach to the design of reconfigurable tree architectures has been presented. The design allocates spares at the leaves of trees and allows sharing of spares between subtrees. The architecture has $O(n)$ VLSI layout for binary trees and is directly extensible to N-ary trees. A lower bound on reliability for a SOFT tree was shown to be more reliable than all modular sparing approaches, with significantly less redundancy. The SOFT architecture is the only known approach to reconfigurable trees which tolerates both link and switch failures. The virtual displacement concept with sharing of spares between clusters of processors is also applicable to other concurrent architectures. The design strategy presented in this paper makes tree architectures attractive for environments where high reliability is required.

REFERENCES

[1] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*. Reading, Mass.: Addison-Wesley, 1980.

[2] S. A. Browning, "A Tree Machine," *Lambda (VLSI Design)*, pp. 31-36, Second Quarter 1980.

[3] S. A. Browning, "Computations on a Tree of Processors," 1979, PhD Thesis, California Institute of Technology.

[4] M. J. Attallah and S. R. Kosaraju, "A Generalized Dictionary Machine for VLSI," *IEEE Trans. on Comput.*, pp. 151-155, February 1985.

[5] J. P. Hayes, "A Graph Model for Fault Tolerant Computing Systems," *IEEE Trans. on Comput.*, pp. 875-883, September 1976.

[6] C. L. Kwan and S. Toida, "An Optimal 2-Fault Tolerant Realization of Symmetric Hierarchical Tree Systems," *Networks* , vol. 12, pp. 231-239, 1982.

[7] C. S. Raghavendra, A. Avizienis, and M. Ercegovac, "Fault Tolerance in Binary Tree Architectures," *IEEE Transactions on Comput.*, pp. 568-572, June 1984.

[8] A. S. M. Hassan and V. K. Agarwal, "A Fault-Tolerant Modular Architecture for Binary Trees Architectures," *Proceedings of the Fifteenth Fault Tolerant Computing Symposium*, pp. 344-349, June 1985.

[9] A. L. Rosenberg, "The Diogenes Approach to Testable Fault Tolerant Arrays of Processors," *IEEE Trans. on Comput.*, pp. 902-910, October 1983.

[10] F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg, "Diogenes: A Methodology for Designing Fault Tolerant VLSI Array Processors," *Proceedings of the Thirteenth Fault Tolerant Computing Symposium*, pp. 23-32, June 1983.

[11] E. Horowitz and A. Zorat, "The Binary Tree as an Interconnection Network: Applications to Multiprocessor Systems and VLSI," *IEEE Trans. on Comput.*, pp. 247-253, April 1981.

[12] J. R. Goodman and C. H. Sequin, "Hypertree: A Multiprocessor Interconnection Topology," *IEEE Trans. on Comput.*, pp. 923-933, December 1981.

[13] A. M. Despain and D. A. Patterson, "X-tree: A Tree Structured Multiprocessor Computer Architecture," *Proceedings of the Fifth Computer Architecture Syposium*, pp. 144-150, April 1978.

[14] C. E. Leiserson, *Area Efficient VLSI Computation*. MIT Press, 1983.

[15] J. B. Ullman, *Computational Aspects of VLSI*. Rockville, Md.: Computer Science Press, 1984.

[16] R. S. Swarz and D. P. Siewiorek, *The Theory and Practice of Reliable System Design*. Bedford, Mass.: Digital Press, 1982.

[17] A. Rosenfeld, "Quadtrees and Pyramids for Pattern Recognition and Image Processing," *Proceedings of the Fifth International Conference on Pattern Recognition*, pp. 802-806, 1981.