Restructuring WSI Hexagonal Processor Arrays

R. Venkateswaran, *Student Member, IEEE*, Pinaki Mazumder, *Member, IEEE*, and Kang G. Shin, *Fellow, IEEE*

***Abstract*—Fault-tolerant approaches have been widely employed to improve the yield of ULSI and WSI processor arrays. In this paper, we propose a host-driven reconfiguration scheme, called HEX-REPAIR, for hexagonal processor arrays characterized by a large number of relatively simple cells. Such arrays have been shown to be the most efficient for many digital signal processing applications, such as matrix multiplication, and for some classes of filtering operations. Reconfiguration for these arrays is made difficult by the asymmetric nature of the interconnection network and the need for keeping the switching overheads at a minimum. The algorithm presented in this paper meets these requirements. In addition, it has excellent fault-coverage characteristics, even in the presence of multiple faults, and can accommodate multiple rows/columns of spare cells. The restructured array is transparent to users and no modification is required in any application program using the array.**

I. Introduction

FOR SOME digital signal processing applications, such as matrix multiplication and some classes of filtering operations, the hexagonal interconnection network has been proposed in literature [2], [3] as the most efficient. When implemented in a wafer-scale integration (WSI) or an ultra-large scale integration (ULSI), a processor array with thousands of cells can be squeezed into a single wafer or chip. One of the main drawbacks with such large integrations is that these arrays are quite prone to defects because of the imperfections in the manufacturing process. A fault not only destroys the regularity of the array, but also may make it useless for the algorithms using the array. Thus fault tolerance is the only solution capable of giving acceptable production yield, as it permits initial testing and subsequent array reconfiguration using spare cells and extra switching hardware. The locality of interconnections and regularity and simplicity of the switching devices are important considerations. Much of the previous work in array reconfiguration [4]–[7] has primarily dealt with rectangular or square arrays. These approaches can be broadly divided into *multiplexer*-based and *switched-bus*-based models.

The *index-mapping* algorithms of [5], [6] fall in the first category. Index mapping refers to the technique of mapping a set of logical indexes onto a set of physical indexes denoting working cells. Depending on the complexity of the actual algorithm, such schemes tend to have a fair survival rate. Each PE is usually directly connected to many other neighboring PE's through large switches/multiplexers capable of connecting four to eight separate parallel buses. A number of extra communication links are needed to attain proper reconfiguration. Clearly such a scheme is justifiable only when the individual PE's are relatively complex, comprising several thousand transistors, thereby making it highly desirable that all the nonfaulty processors be incorporated into the working logical array.

On the other hand, the switched-bus architecture, as employed in [7] and others, is quite attractive when the area for an individual cell is small. The reconfiguration is based on a fixed number of horizontal and vertical buses connected using simple 2×2 switches. A reconfiguration technique called *modified* indexed mapping is used to overcome the deterministic process of index modification. Reconfiguration is analyzed with respect to the switching and routing capabilities of the interconnection network. However, the generalization of these schemes to apply to hexagonal arrays has not been studied much. The task is certainly nontrivial, owing to the asymmetric nature of the hexagonal interconnection and having to maintain *transparent* connections to all six logical neighbors for each PE.

Gordon *et al.* [11] proposed the first reconfiguration scheme intended primarily for hexagonal arrays wherein the individual processors occupy relatively small silicon area. It worked by bypassing faulty as well as some fault-free PE's, using a bare minimum in terms of extra switch hardware or links. However, the approach suffers from very poor fault coverage and processor utilization in the event of multiple faults. The authors justify it by claiming that the probability of getting a chip with only one or two faults is quite high. The reconfiguration algorithm HEX-REPAIR, presented in this paper, is based on similar assumptions as [11]. However, HEX-REPAIR is much more robust and has fault coverage rates comparable to the index-mapped schemes. The rest of the paper is organized as follows. In Section II, we present the terminology and the fault model used. Sections III and IV explain in detail the inner workings of the algorithm. The proof of correctness is derived in Section V, while the fault-coverage characteristics are studied in Section VI. Section VII pertains to implementation related aspects such as the area and delay penalty of reconfiguration.

Manuscript received April 18, 1990; revised December 31, 1992. This work was supported by the Army Research Office under URI Program Grant DAAL 03-87-K-0007, by the Office of Naval Research under Grant 00014-85-K-0122, and by the National Science Foundation under Grant 9013092. This paper was recommended by Associate Editor F. Brglez.

The authors are with the Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor, MI 48109-2122.

IEEE Log Number 9200908.

II. Preliminaries

Before embarking on the mechanics of HEX-REPAIR, it will be illuminating to understand the assumptions made.

a) PE's, both faulty and nonfaulty, can be bypassed during reconfiguration using simple switches. The bypassed cells will be referred to as *switching elements* (SE) to indicate the fact that they cease to perform processing, but serve to maintain the logical interconnections between logically adjacent cells. The switching-elements that are actually working processors are sometimes called *pseudo-faults* in the literature.

b) A conventional assumption that many approaches [16], [9], [5,] [15] make is that faults affect only the PE's. The switches and interconnects are considered to be fault-free. We also make this assumption. The rationale here is that the PE's have to be designed with leading edge rules so as to maximize speed and density, while the switches and interconnects can be designed with more conservative design rules. Also, interconnects are often formed on a single layer, whereas the PE's use multiple layers. This leads to additional failure modes for the processors such as mask misalignment and interlayer shorts, thereby reducing the chip yield as a polynomial function of the number of masking levels used.

c) Each PE is associated with a six-port switch that can provide six different switching functions. Such a switch can be designed with as few as 15 ON/OFF devices. This is the only overhead in terms of extra hardware that has to be incorporated into the array. Such low overhead is critical, especially for digital signal processing applications where the processors may only comprise a few hundred transistors. Using multiplexers, the relative switching hardware overhead per processor increases to an extent that the validity of the fault-free switch fault model becomes questionable.

d) Our approach is intended for off-line reconfiguration, mainly to handle production failures. Testing information regarding the location of faults is assumed to be available using schemes such as the one suggested in [17].

The goal of reconfiguration is to recover a logical hexagonally connected array of working cells from the original physical array. Each cell of the array is represented by a pair of *physical* indexes (p_x, p_y), and a pair of *logical* indexes (l_x, l_y) that indicate the indexes of the function of each cell at runtime. The two are the same in the absence of faults, but can differ when faults occur. Logical indexes of faulty cells and the SE's are set to zero. We let $\mathcal{F}$ denote the fault set, i.e., the collection of physical indexes of the faulty elements in the array; and let $|\mathcal{F}|$ denote the size of $\mathcal{F}$. Further, we let X_i and Y_i denote the p_x and p_y values of the ith fault, respectively.

Definition 1: Two cells are said to be *horizontally* connected if they lie on the same physical row. *Vertically* and *diagonally*[1] connected cells are defined similarly.

[1]By *diagonal*, we refer to the one going from the top left of the array and proceeding to the bottom right.

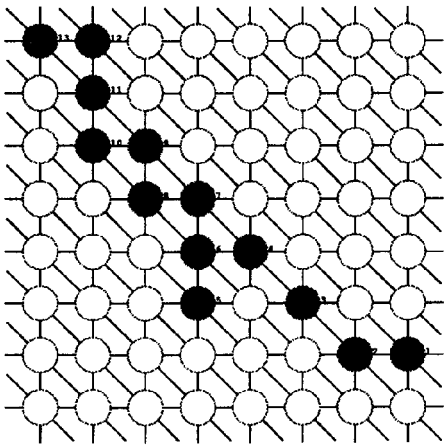


Fig. 1. An 8 × 8 hexagonal array with random faults.

Definition 2: Two cells are said to be *row* (*column*) connected if they are either horizontally (vertically) or diagonally connected.

Definition 3: An *H line* is any collection of n' row-connected PE's, where n' is the number of columns in the physical grid. Thus, each H line starts at column 1 and ends at the last column. Similarly, a *V line* is a collection of m' column-connected PE's, where m' is the number of rows in the physical grid.

The algorithm HEX-REPAIR, presented in this paper, can be divided into the following two phases:

Covering Phase: This determines an appropriate set, if one exists, of H and V lines that cover all faulty cells. All PE's on these cover lines are treated as switching elements in the final solution.

Procedure 1: The Covering Phase

1) *Fault Enumeration:* Find a one-to-one mapping Φ: $\mathcal{F} \rightarrow \{1, 2, \cdots, |\mathcal{F}|\}$.
2) *Graph Construction:* Construct the HCG and the VCG for the given fault pattern.
3) *Graph Transversal:* Determine the set S_h of all distinct paths in the HCG and the set S_v of all distinct paths in the VCG that start at a *root* node and end at the *sink* node.
4) *Integer Programming:* Determine the solution-set $\mathcal{S}$ consisting of n_h paths from S_h and n_v paths from S_v, which together cover all faults in $\mathcal{F}$ and which satisfy

$$n_h \leq \text{number of spare rows} = R$$

$$n_v \leq \text{number of spare columns} = C.$$

Configuration Phase: Here we configure each SE so as to ensure proper interconnections in the reconfigured array. A simple scheme is presented to determine the particular configuration of an SE, based on the nature of interaction between the H lines and V lines at that array

location. The final step is in assigning the new logical indices.

Fig. 1 shows an 8 × 8 physical grid. The indexes of the 13 shaded processing elements represent the fault set $\mathcal{F}$. This example array can be reconfigured using just one spare row and one spare column of cells to yield a target logical array of dimension 7 × 7. We shall use this problem as our running example throughout the rest of this paper for illustrative purposes.

III. The Covering Phase

In addition to finding a suitable set of H and V lines, this phase also determines the maximum size working logical array that can be recovered from the given (faulty) physical array. If this maximum is deemed to be insufficient, the reconfiguration process can be either terminated or more spares can be added and the algorithm repeated. Procedure 1 outlines the main steps involved in the covering phase. These are explained in greater detail in the following paragraphs.

A. Fault Mapping

Let D_j be the set consisting of cells with physical indexes (x, y) such that $j = x + y - 1$. Initialize the starting *fault index* to $|\mathcal{F}.|$ Now, visit the cells in each set $D_1, D_2, \cdots$ in the increasing order of y for each D_j. For example, in Fig. 2, the processing elements are visited in the order: (1,1), (2,1), (1,2), (3,1), (2,2), (1,3), (3,2), (2,3), (1,4), (3,3), (2,4), (3,4). As each fault is encountered, it is mapped to the current *fault-index* value and the *fault-index* value is then decremented by one. The fault indexes for the example problem are indicated besides the faulty cells in Fig. 1.

This step thus assigns a number between 1 and $|\mathcal{F}|$ to each fault in the array. This mapping is used in the construction of the cover graphs and H/V lines.

B. Horizontal and Vertical Cover Graphs

The horizontal and vertical cover graphs (HCG), (VCG) aid in identifying the maximal sets of faults such that all the faults in one set can be simultaneously covered with a single spare row (HCG) or column (VCG). The idea is to impose a partial order on the set $\mathcal{F}$ so that the search for suitable lines can be restricted to just the fault set, rather than the whole physical array. This is especially important for large arrays with only a few faults.

Formally, these graphs can be defined as follows:

Definition 4: The HCG for a given $\mathcal{F}$ consists of $|\mathcal{F}| + 1$ nodes, one for each fault and a special *sink* node. There exists a directed edge from node i to node j, iff

$Y_i > Y_j$ and $X_i \geq X_j$ and $Y_i - Y_j \geq X_i - X_j$;
j is the *sink* node,
there do not exist nodes $k, i_1, i_2, \cdots i_n$ in HCG such that $i \rightarrow k$ and $k \rightarrow i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_n \rightarrow j$. In other words, i should not be directly connected to a node that is an ancestor of j.

Definition 5: The VCG for a given $\mathcal{F}$ consists of $|\mathcal{F}| + 1$ nodes, one for each fault and a special *sink* node. There exists a directed edge from node i to node j, iff

$X_i > X_j$ and $Y_i \geq Y_j$ and $X_i - X_j \geq Y_i - Y_j$;
j *is the sink* node;
There do not exist nodes $k, i_1, i_2, \cdots i_n$ in VCG such that $i \rightarrow k$ and $k \rightarrow i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_n \rightarrow j$.

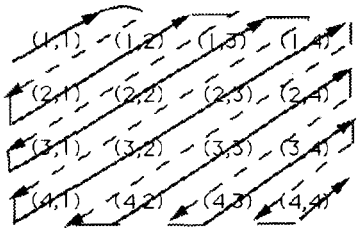
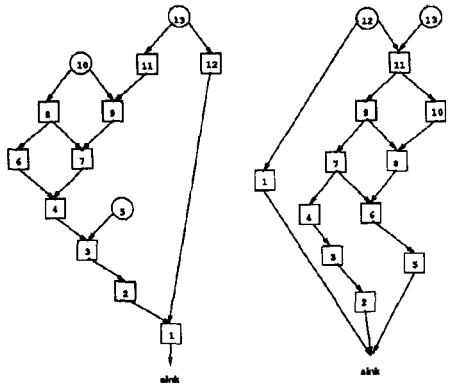


Fig. 2. Enumeration scheme for determining *fault indexes*.



(a) (b)

Fig. 3. (a) Horizontal cover graph. (b) Vertical cover graph.

The HCG and VCG, as defined above, can be constructed in $O(|\mathcal{F}|^2)$ time. Fig. 3 shows the two 13-node HCG and VCG for our running example. The circled nodes are those with no ancestors in the graph. These will be referred to as *root* nodes.

C. Graph Traversal

The set $S_h(S_v)$ is a collection of paths in the HCG (VCG, respectively), starting from a *root* node and terminating at the *sink* node. New paths are generated using a depth-first search strategy. Since, theoretically, the number of possible paths in each graph could be as high as $2^{|\mathcal{F}|/2}$, we use a parameter, called *MaxPathLimit*, to restrict the maximum number of paths generated for each graph. Procedure 2 outlines the path traversal method.

The value set for *MathPathLimit* represents a tradeoff between computation time and a 100% guarantee of finding a feasible solution if one exists (*MaxPathLimit* = infinity). In our simulations, we have set *MaxPathLimit* to 1000 because we assume that the faults occur randomly and the number of possible paths in each graph is relatively small. Thus, the probability that a fault (node in the graph) will occur in one of the 1000 chosen paths is quite high.

Procedure 2: Path Traversal Phase

```
PathsGenerated = 0;
RootList = a circular list of all the root nodes in HCG (VCG);
While (PathsGenerated < MaxPathLimit) and (RootList not empty) do
        CurrentRoot = Next entry in the RootList;
        Generate a new path from CurrentRoot to the sink node;
        If no new path exists then
              Remove CurrentRoot from the RootList
        Else
              Increase PathsGenerated by one; Add this path to S_h (S_v)
        Endif
EndWhile
```

D. Finding the Solution Set $\mathcal{S}$

The solution set $\mathcal{S}$ comprises of n_h paths from S_h and n_v paths from S_v, such that together they cover all the faults in $\mathcal{F}$, where $n_h(n_v)$ is less than or equal to the number of spare rows (spare columns) available. This problem is similar to the prime implicant covering problem commonly encountered in gate minimization. A natural representation is the *cover table* that contains a row for each horizontal path (member of S_h) and for each vertical path (member of S_v) and a column for each fault in $\mathcal{F}$. A ✓ is placed in the ith row and jth column if fault j is covered by the ith (horizontal and vertical) path. Certain rows and columns can be deleted from the *cover table* to simplify determining $\mathcal{S}$. The rules for simplification follow.

1) Suppose, for a given column j, a ✓ occurs only once in row i. Then this row can be considered an *essential* path and a part of every solution. This is because fault j can only be covered and replaced by this path. All the other columns k for which there is a ✓ in row i can also be removed.
2) If two or more columns are identical, all but one can be deleted.
3) Row i is said to *dominate* row j if it has a ✓ in each column where j has one, and also in a few additional columns. Furthermore, both i and j should either be horizontal paths or both be vertical paths. In this case, the dominated row j can be deleted.
4) Column i is said to *dominate* column j if i contains a ✓ in every row where j has a ✓, and in addition i has some more ✓ in other rows too. In such a case, we can remove the dominating column.

Hence, if we let B_i be a Boolean variable that is 1 if row i of the *cover table* is selected for inclusion in $\mathcal{S}$, and $B_{j1}, B_{j2} \cdots B_{jn_j}$ be the rows that cover column (fault) j; then the solution for covering all faults is given by the product-of-sums Boolean equation

$$\prod_{j=1}^{n} (B_{j1} + B_{j2} + \cdots + B_{jn_j}) = 1.$$

Using the distributive laws of Boolean algebra, this can be equivalently represented by a sum-of-products expression of the form

$$\sum_{k=1}^{p} (B_{k1} B_{k2} \cdots B_{kn_k}) = 1.$$

A product term with the fewest number of literals will constitute the minimum number of spares that are needed to obtain a reconfiguration solution. However, any product term that consists of fewer than n_h horizontal paths and n_v vertical paths, is an acceptable candidate for reconfiguration.

Example: Table I is the *cover table* for the reconfiguration problem shown in Fig. 1. The top half of the table indicates the six horizontal paths that exist in the HCG, while the latter half gives the nine vertical paths that are present in the VCG. Furthermore, columns 2 and 3 can be removed, as they dominate column 4. A solution for this coverage is easily determined to be $\{A, G\}$ containing one horizontal path (A) and one vertical path (G). So it is admissible, as we do have one spare row and one spare column.

In practice, we solve the covering problem by employing the *integer linear program* paradigm for which efficient computer routines are available [18]. Let R and C be positive integers denoting the number of available spare rows and columns. Let each horizontal path be denoted by a variable x_i and let each vertical path be denoted by a variable y_i. Then our integer program can be stated as

Find a set of values for the m integer variables $x_1, x_2, \cdots, x_m$ *and the n integer variables* $y_1, y_2, \cdots, y_n$, *which minimize the objective function:*

$$R \cdot \sum_{i=1}^{m} x_i + C \cdot \sum_{i=1}^{n} y_i$$

Subject to the $|\mathcal{F}| + 2$ *constraints of the form*

$$\sum_{i=1}^{m} a_{i,j} x_i + \sum_{i=1}^{n} a_{(i+m),j} y_i \geq 1,$$

$$\text{where } j = 1, 2, \cdots, |\mathcal{F}|;$$

$$\sum_{i=1}^{m} x_i \leq R; \quad \sum_{i=1}^{n} y_i \leq C$$

TABLE I
FAULT COVER TABLE

Path name	Fault Index 1	2	3	4	5	6	7	8	9	10	11	12	13
A	✓	✓	✓	✓			✓		✓		✓		✓
B	✓	✓	✓	✓			✓		✓	✓			
C	✓	✓	✓	✓			✓	✓		✓			
D	✓	✓	✓	✓		✓		✓		✓			
E	✓	✓	✓		✓								
F	✓											✓	✓
G					✓	✓		✓		✓	✓	✓	
H					✓	✓		✓		✓	✓		✓
I					✓	✓		✓	✓		✓	✓	
J					✓	✓		✓	✓		✓		✓
K					✓	✓	✓		✓		✓	✓	
L					✓	✓	✓		✓		✓		✓
M		✓	✓	✓			✓		✓		✓	✓	
N		✓	✓	✓			✓		✓		✓		✓
O	✓											✓	

Here the quantities $a_{i,j}$ are either 0 or 1 and are obtained from the cover table. The solution set $\mathcal{S}$ includes all paths that correspond to a nonzero x_i or y_i value at the end of the optimization. The constraints ensure that the number of horizontal paths chosen do not exceed the number of spare rows and the number of vertical paths do not exceed the number of spare columns. The first set of $|\mathcal{F}|$ constraints ensure that each fault is accounted for in at least one member of $\mathcal{S}$.

For m rows and n columns, the algorithm typically finds an reach the optimal vector after a number of iterations that is no bigger than the order of m or n, whichever is larger. In our case, m is determined by the value for *MaxPathLimit* and is independent of the array or fault-set size and $n = |\mathcal{F}|$. Thus, the number of iterations required is solely determined by the number of faults. For most arrays with up to 25–30 faults that we simulated, we obtained a solution in less than a second working on a Sun Sparcstation. It is interesting to note that network flow techniques can also be used to determine $\mathcal{S}$.

IV. THE CONFIGURATION PHASE

Once the horizontal and vertical covering paths have been successfully determined, the configuration phase is next invoked. The three main steps constituting this phase are stated in Procedure 3.

Procedure 3: The Configuration Phase

1) For each horizontal (vertical) path p in $\mathcal{S}$, construct a nonoverlapping H line (V line) that includes all faults in p.
2) Reconfigure the array by assigning logical indices to the remaining good processing elements.
3) Configure all the SE appropriately so as to ensure that the restructured array is transparent to the various alogrithms using the hexagonal array. Thus a PE can continue to communicate with its logical neighbors on the same links as before, unaware that it now reaches them through some SE's.
4) Perform postprocessing to shorten logical connections.

A. Constructing H and V Lines

Each horizontal path in $\mathcal{S}$ is to be covered by an H line and each vertical path by a V line. All of the cells of these lines will become SE's in the final solution. With each of these SE's, we associate an *H type* and a *V type* based on the manner in which the H/V lines pass through it (see Fig. 4).

Based on the H line l passing through an SE with indexes (i, j), its H type can be classified as being

1) U: (Unset) No H line passes through (i, j), which means that this SE is intersected only by a V line.
2) S: (Straight) l connects (i, j) to $(i, j - 1)$ and $(i, j + 1)$, if they exist.
3) D: (Diagonal) l connects (i, j) to $(i - 1, j - 1)$ and $(i + 1, j + 1)$.
4) LB: (Lower Bend) l connects (i, j) to $(i - 1, j - 1)$ and $(i, j + 1)$.
5) UB: (Upper Bend) l connects (i, j) to $(i + 1, j + 1)$ and $(i, j - 1)$.
6) TR: (TwoRow) This is the case when two H lines meet at (i, j).

Likewise, SE (i, j) can have its V type set to one of the following six categories.

1) U: (Unset) No V line passes through (i, j).
2) S: (Straight) l connects (i, j) to $(i, -1, j)$ and $(i + 1, j)$, if they exist.
3) D: (Diagonal) l connects (i, j) to $(i - 1, j - 1)$ and $(i + 1, j + 1)$.
4) LB: (Lower Bend) l connects (i, j) to $(i - 1, j)$ and $(i + 1, j + 1)$.
5) UB: (Upper Bend) l connects (i, j) to $(i - 1, j - 1)$ and $(i + 1, j)$.
6) TC: (TwoCol) This is the case when two V lines meet at (i, j).

The initial strategy used to construct an H/V line for each path is stated below. "Let the horizontal (vertical) path p in $\mathcal{S}$ consist of P faults $\{f_1, f_2, \cdots f_P\}$ arranged in decreasing order of their fault indexes. The fault f_1 is horizontally (vertically) connected to the PE in the first column (row) of the row (column) of f_1. Similarly, f_P is horizontally (vertically) connected to the PE in the last column (row) of the row (column) of f_P. For any other two faults f_i and f_{i-1}, we find a (unique) PE k which can be horizontally (vertically) connected to f_i and diagonally connected to f_{i+1}."

In the presence of faults occurring in more than one line, this procedure can lead to some overlap between H/V lines. We remove such overlap by removing the common faults from all but one line and reconstruct the other lines as before.

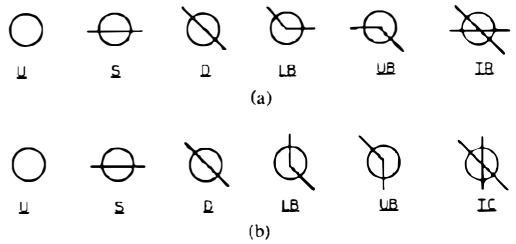


Fig. 4. Different H and V line patterns. (a) H line types. (b) V line types.

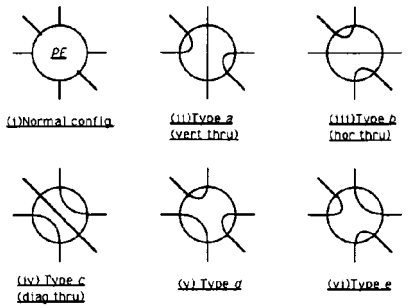


Fig. 5. Different SE configurations

B. Configuring the Switching Elements

A byproduct of the covering strategy used is that the configuration of the SE's to maintain logical transparency is reduced to a simple TABLE-LOOKUP operation. Note that when we refer to an SE configuration, we actually mean the configuration of the switch associated with that PE. Fig. 5 shows the six different switching functions, of which the first corresponds to a good PE and the latter five correspond to an SE of type "a," "b," "c," "d," and "e," respectively. Table II is the SE configuration table. Entries marked with a "-" indicate invalid H type/V type combinations.

C. Assignment of Logical Indexes

Procedure 4 describes the scheme for assigning the logical indices to the working cells of the reconfigured array. First we scan, column by column, assigning a logical row index to each PE in that column. This value is determined by the number of H lines that traverse the column above the row under consideration. An SE gets a logical row index of 0.

We then perform a similar operation on each row, assigning a logical column index to each PE, based on the number of V lines met until that point. It is possible for two cells, C_1 and C_2, to get assigned the same logical indices when either two H lines or two V lines intersect at the same SE. However, we note that for an $n \times n$ physical array, the total number of SE's that result from two such intersecting H or V lines is equal to $2n - 1$. On the other hand, every spare row or column consists of n cells. Thus, we end up with an extra spare cell. We can therefore resolve the conflict by converting one of C_1 or C_2 to an SE.

Corollary 1: Each switching element in the reconfigured array is intersected by at most one H line and at most one V line. The proof follows from the H–V line construction and conflict resolution strategy employed in Procedure 4.

Fig. 6 is the final result for our running example. The *logical* indexes of all the working cells were determined using Procedure 4, and the SE's were configured as per Table II.

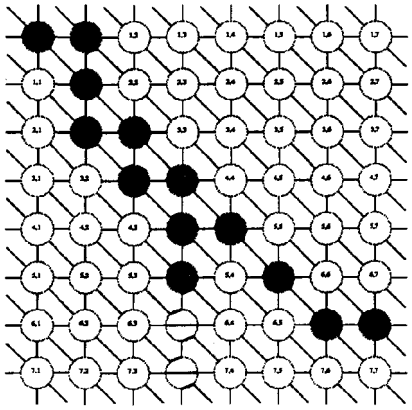


Fig. 6. Final result of the reconfiguration.

TABLE II
LOOKUP TABLE FOR SE SETTINGS

H type → / V type ↓	U	S	D	LB	UB	TR
U	—	a	a	a	a	a
S	b	c	a	e	d	c
D	b	b	—	—	—	—
LB	b	e	—	e	—	—
UB	b	d	—	—	d	—
TC	b	c	—	—	—	—

Procedure 4: Assignment of New Logical Indexes

```
For each column c Do
    d_R = 0 (d_R counts the number of H lines met so far)
     For each row r Do
         If (r, c) is a SE Then
            d_R = d_R + k (where k = 2 if H line state for (r, c) is TR, 0 if it is U,
            and 1 otherwise)
         Else
         If (r, c) is free Then
            Mark logical row for (r, c) as r - d_R
```

```
End For
  For each row r Do
    d_C = 0 (d_C counts the number of V lines met so far)
     For each col c Do
         If (r, c) is a SE Then
            d_C = d_C + k (where k = 2 if V line state for (r, c) is TC, 0 if it is U,
            and 1 otherwise)
         Else
         If (r, c) is free Then
            Mark logical col for (r, c) as c - d_C
            Conflict Resolution:
            If a PE (i, j) has already been assigned the same
            logical indices Then
              If |r - i| ≥ |c - j| Then
                 Mark (i, j) as an SE of type a
              Else
                 Mark (i, j) as an SE of type b
End For
```

V. Proof of Correctness of the Reconfiguration Algorithm

Conceptually, the various H and V lines partition the array into several subareas, each possessing a certain relationship between the logical and physical indices for the processing elements within. In our example, there are three such subareas: A, B, and C, (described in Table III). We generalize this concept to several H and V lines, thereby giving a constructive proof of the correctness of HEX-REPAIR.

Theorem 1: The LOOKUP TABLE for switching element settings guarantees a proper hexagonally connected logic array of working PE's after the Logical Index Assignment phase.

Proof: Let $T(r, c)$ denote a subarea of working cells which is bounded by r H lines above and c V lines to the left. Then in this subarea, the logical indices of any cell C_1, with physical indexes (i, j), is given by $(i - r, j - c)$. The proof of the correctness of the algorithm can be established by enumerating the various possible cases.

Consider the logical right neighbor of C_1. If the cell $(i, j + 1)$ is good, then we are done. Assume otherwise. There are two cases to be examined here. Let C_2 be the first cell on the same row as C_1 and to its right, that is either free or contains an H line. Let the physical indices of C_2 be $(i, j + k)$. Consequently, all cells between C_1 and C_2 have a V line passing through them and will become SE's of type "b."

Case 1: C_2 is a free, i.e., working PE.

In this case, we have $(i, j)_R \rightarrow (i, j + 1)_L \rightarrow (i, j + 1)_R \rightarrow \cdots \rightarrow (i, j + k)_L$. The subscripts denote the port of entry or exit of that cell, viz. Left, Right, Top, Bottom, Top Diagonal, and Bottom Diagonal. Since there are $k - 1$ V lines between C_1 and C_2, the latter lies in a subarea of type $T(r, c + k - 1)$. Thus, the logical indexes of C_2 are $(i - r, (j + k) - (c + k - 1)) = (i - r, j - c + 1)$, which are the logical indexes required for the right neighbor of C_1.

Case 2: C_2 contains an H line.

By similar arguments to the previous case, we can conclude that the right neighbor link of C_1 reaches the left input of C_2. Furthermore, the H line type of C_2 can only be *Diagonal* or *Lower Bend*. This is because otherwise, cell $(i, j + k - 1)$ will also have an H line, which contradicts the assumption that C_2 is the first cell containing one. Also cell $C_1' = (i - 1, j + k - 1)$ is diagonally connected to C_2 via the same H line. So the H line type of C_1' can only be *Diagonal* or *Upper Bend.*

Similarly, a *V* line, if any, through C_2 can be *Straight* or *Lower Bend*; and a V line, if any, through C_1' can only be *Straight* or *Upper Bend*. From Table II, C_2 can therefore be of type "a" or "e"; and C_1' can be of type "a" or "d." In all cases, the signal path arriving at the left port of C_2 goes to the bottom diagonal port of C_1' and then comes out of its right port.

Since at most one H line can pass through any cell, C_1' cannot be simultaneously horizontally connected to cell $(i - 1, j + k)$ by an H line. Thus, we can extend the same arguments we made before, by replacing occurrences of C_1 and C_1'. We observe that with each H line, we move up one physical row; and with each V line we proceed one column to the right. Let this process finally terminate at a good cell C_g which lies in a subarea of type $T(r - r', c + c')$. This means that the signal path starting at the right port of C_1 meets r' H lines and c' V lines. Therefore, the physical indexes of C_g will be $(i - r', j + c' + 1)$. Thus, the logical indices of C_g are $((i - r') - (r - r'), (j + c' + 1) - (c + c')) = (i - r, j - c + 1)$, which is as required.

The proof of correctness for up/down neighbor connections can be similarly derived. The case of diagonal connections is slightly more involved. If the cell $(i + 1, j + 1)$ has a *Straight* V line, then the path traced will be $(i, j)_{DR} \rightarrow (i + 1, j + 1)_{DT} \rightarrow (i + 1, j + 1)_T \rightarrow (i, j + 1)_B \rightarrow (i, j + 1)_{DB} \rightarrow (i + 1, j + 2)_{DT}$. If on the other hand,

TABLE III
PARTITIONING DUE TO H AND V LINES

	Phy Index		Log Index	
SubArea	x	y	x	y
A	i	j	i − 1	j
B	i	j	i	j − 1
C	i	j	i − 1	j − 1

the cell $(i + 1, j + 1)$ has a *Straight* H line, then the path traced will be $(i, j)_{DB} \rightarrow (i + 1, j + 1)_{DT} \rightarrow (i + 1, j + 1)_L \rightarrow (i + 1, j)_R \rightarrow (i + 1, j)_{DB} \rightarrow (i + 2, j)_{DT}$. Finally, the cell $(i + 1, j + 1)$ can have both a *Straight* V line and a *Straight* H line. In this case the path traced will be $(i, j)_{DB} \rightarrow (i + 1, j + 1)_{DT} \rightarrow (i + 1, j + 1)_{DB} \rightarrow (i + 2, j + 2)_{DT}$. Hence, if the path encounters r' *H* lines and c' V lines, the physical coordinates of C_g are $(i + r' + 1, j + c' + 1)$. Since the type of subarea containing C_g is given by $T(r + r', c + c')$, the logical indexes of C_g are $((i + r' + 1) - (r + r'), (j + c' + 1) - (c + c')) = (i - r + 1, j - c + 1)$, which are as required. Q.E.D.

VI. EVALUATION OF THE ALGORITHM

Reconfiguration algorithms, such as HEX-REPAIR, can be compared by their probability of getting stuck in a fatal situation, namely, one for which the algorithm fails to find a suitable reconfiguration. Even for optimal algorithms based on local connections only, there are small clusters of faults that cannot be overcome. For fixed probability of cell failure, the probability that such clusters occur therefore approaches one as the array becomes infinitely large. Thus, there is a critical size of the array that cannot be overcome unless the probability of single-cell failure decreases. Thus, in such cases, a degradation in array size would result unless new spares are added.

One pertinent measure is the critical constant α_c which is defined [19] to be the largest number such that if $\alpha < \alpha_c$, then for an array with N processors and individual processor failure probability of $1/N^{\alpha}$, the reconfiguration will almost certainly get stuck in a fatal situation. Thus, it is desirable to have as small a value for α_c as possible. The rest of this section pertains to deriving the critical constant for HEX-REPAIR.

Definition: An *atomic* fail pattern is defined as a fail pattern that cannot be solved by the algorithm; while removing any one fault from the pattern leads to a reconfigurable array.

Theorem 2: For any array of size $n \times n$ with $N = n^2$ processors, HEX-REPAIR contains $O(n^6) = O(N^3)$ different fail patterns of size 3.

Proof: An atomic fail pattern for HEX-REPAIR is as follows: Pick any cell other than one lying along the top two rows of the two rightmost columns. Place the first fault there. Now place the second fault to the right and above the first fault. Likewise, place the third fault to the right and above the second fault. Fig. 7 shows a typical member of the fail-pattern family. It consists of patterns

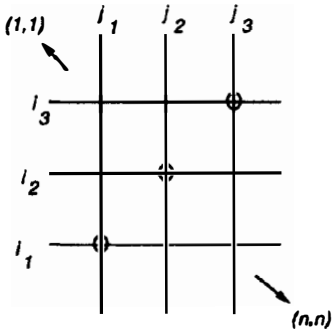


Fig. 7. Typical member of an atomic fail-pattern for α.

of the form:

$$\{(i_1, j_1), (i_2, j_2), (i_3, j_3)\}, \qquad i_1 > i_2 > i_3; j_1 < j_2 < j_3$$

where the top left cell is labeled (1, 1).

Now, consider any three columns. It is clear that the number of patterns T satisfying the constraints mentioned above is given by

$$T = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} 1 = n(n-1)(n-2)/6 = \binom{n}{3}.$$

We can also choose the columns in $\binom{n}{3}$ ways. Hence, the total number of patterns for such an array is $\binom{n}{3}^2$. In asymptotic notations[2] this implies there are $O(n^6) = O(N^3)$ different fail patterns of size 3. QED

Theorem 3: The critical constant α_c for the reconfiguration algorithm HEX-REPAIR approaches 1 as the number of processors N tends to infinity.

Proof: We consider only the case when all fail patterns A are each the size k. This is true for HEX-REPAIR. Let F_A be a random variable equal to the number of fail patterns in the processor array G. Let $E(F_A)$ and $E(F_A^2)$ be its first and second moments, respectively. Then it was shown in [19] that for positive constants a and β

$$E(F_A) = a \cdot N^{\beta - k\alpha}$$

$$E(F_A^2) = E(F_A) + T_{A,N} \cdot \sum_{i=0}^{k-1} (S(i) \cdot p^{2k-i})$$

where $T_{A,N}$ is the number of fail patterns in G; $S(i)$ denotes the number of fail patterns B that have exactly i common faults with any given fail pattern A; and $N^{\beta} \propto T_{A,N}$.

For one spare row and one spare column, by Theorem 2, each fault pattern is of size $k = 3$. Hence, β is also 3. Therefore,

$$E(F_A) = a \cdot N^{3(1-\alpha)}. \tag{6.1}$$

If $\alpha > 1$, then clearly $E(F_A) \rightarrow 0$ as $N \rightarrow \infty$. Thus, there is almost certainly no fail pattern in G. This proves the first part of the theorem. Likewise, if $\alpha < 1$, then $E(F_A) \rightarrow \infty$ as $N \rightarrow \infty$. In this case, to establish that $\Pr(F_A$

[2] $O(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

[3] $o(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

$= 0$) as $N \to \infty$, we have to consider Chebyshev's inequality. This implies for $E(F_A) \neq 0$

$$\Pr(F_A = 0) \leq \frac{E(F_A^2)}{E(F_A)^2} - 1.$$

It can be further shown that for each fail pattern A of HEX-REPAIR,

$$S(0) = a^2N^3 - o(N^3),\ S(1) = O(N^2)$$
$$S(2) = O(N).$$

Hence, for our fail pattern family, with the individual processor failure probability of $p = N^{-\alpha}$, it follows that

$$\begin{aligned} E(F_A^2) &= E(F_A) + N^3 \cdot (a^2N^3 - o(N^3)) \cdot p^6 \\ &\quad + O(N^3 \cdot n^4 \cdot p^5 + N^3 \cdot n^2 \cdot p^4) \\ &= E(F_A) + E(F_A)^2 - o(E(F_A)^2) \\ &\quad + O(N^{5(1-5\alpha)} + N^{4(1-\alpha)}). \end{aligned}$$

Hence, dividing by $E(F_A)^2 = a^2 \cdot N^{6(1-\alpha)}$, we get

$$\frac{E(F_A^2)}{E(F_A)^2} - 1 = \frac{1}{E(F_A)} - o(1) + O(N^{\alpha-1} + N^{2(\alpha-1)}).$$

As $N \to \infty$, the right hand side $\to 0$. Thus, from the Chebyshev inequality, we can conclude that $Pr(F_A = 0) \to 0$ as $N \to \infty$ for $\alpha < 1$. Q.E.D.

The same treatment can be extended to the case of say r spare rows and c spare columns. The *atomic* fault pattern now contains $k = r + c + 1$ faults and is constructed similar to the 3-fault pattern. Each successive fault occurs to the top and right of the preceding one. The number of such patterns is given by the expression:

$$\binom{n}{k} \cdot \sum_{i_k=1}^{u_k} \sum_{i_{k-1}=1}^{u_{k-1}} \cdots \sum_{i_2=1}^{u_2} \sum_{i_1=1}^{u_1} 1 = \binom{n}{k}^2$$

where the summation in each case is performed until

$$u_j = (n - j + 1) - \sum_{r=(j+1)}^{k} i_r$$
$$u_k = (n - k + 1).$$

When $N \to \infty$, i.e. when $N >> k$, the number of patterns is $O(N^k)$. Since the number of faults in each pattern is k, we get the critical constant $\alpha < 1$.

Thus, to summarize, our algorithm guarantees a reconfigurable solution either when 1) the number of faults $<$ the number of spare rows and columns, or 2) the probability of failure of an individual processing element $\leq 1/N$, where N is the number of preoccessors in the array.

The analysis of Theorem 3 has been verified by extensive simulations. Fig. 8 shows the successful reconfiguration rate (100% fault coverage), and Fig. 9 the average number of faults that could be covered. The data points used to plot the graphs were the average taken over 1000 independent runs using one spare row and one spare column of cells. The dimension of the arrays used is indicated in the legend. Especially for the larger arrays, we achieve 100% fault coverage when $\alpha > 1$ as expected.

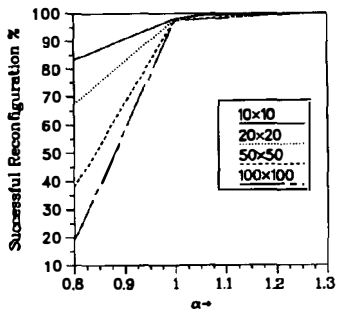


Fig. 8. Effect of α on reconfiguration.

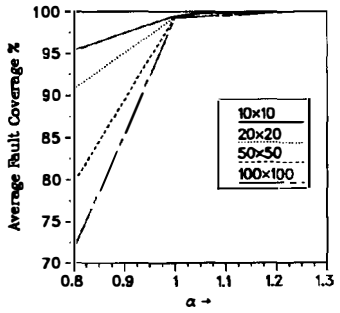


Fig. 9. Effect of α on average fault coverage.

VII. Implementation Issues

A switch which realizes the six different switching functions needed in HEX-REPAIR can be realized using only 15 basic ON/OFF devices per cell. This is shown in Fig. 10. Note, no additional datapaths or multiplexers or long switched buses are required. Table IV shows how the different transistors need to be set ON and OFF in the six cases. Thus, for the hexagonal array, this scheme entails only 2.5 switching devices per port of a cell.

On the other hand, the switching overheads for direct-reconfiguration and other types of fault-stealing approaches [5], [6] is quite high. Based on the reconfiguration rules for the direct scheme, for any cell (i, j), possible logical neighbors are:

- along vertical axis: cells $(i-2, j-1)$, $(i-1, j-1)$, $(i, j-1)$, $(i-1, j)$, $(i-2, j+1)$, $(i-1, j+1)$, $(i, j+1)$.
- along the horizontal axis: cells $(i-1, j-1)$, $i, j-1$), and $(i+1, j+1)$.
- along diagonal: cells $(i+1, j)$, $(i-1, j)$, $(i-2, j)$, $(i-3, j)$, $(i, j-1)$, $(i-1, j-1)$, $(i-2, j-1)$, $(i+1, j-2)$, $(i, j-2)$, $(i-1, j-2)$, $(i-2, j-2)$, $(i-3, j-2)$.

Thus, each cell needs a 7×1, a 3×1 and a 12×1 mux for selecting inputs and a 2×1 mux for selecting

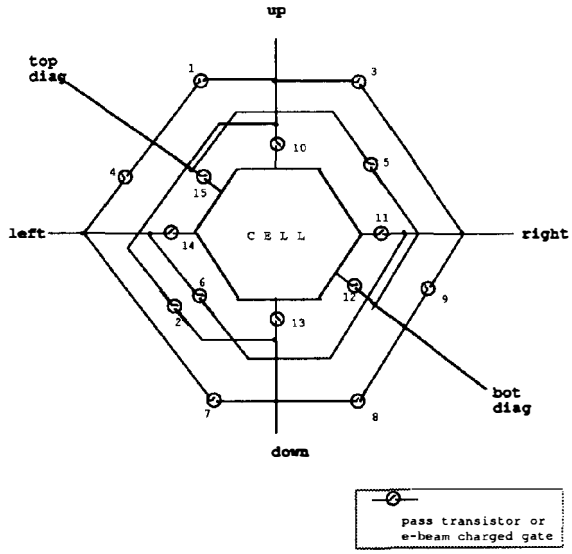


Fig. 10. Switch design using pass transistors or floating gate transistors.

TABLE IV
SWITCHING FUNCTION TABLE

Switch Configuration	•N/OFF settings														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Normal cell	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
SwType *a*	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0
SwType *b*	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0
SwType *c*	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0
SwType *d*	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0
SwType *e*	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0

outputs. Direct interconnections are also required between cells with a Manhattan distance of 5. The total length of interconnect required thus is considerable.

For production-time reconfiguration, these devices are ideally e-beam/UV programmed floating gate transistors. By e-beam addition or UV deletion of the gate charge, the transistor can be set to either the ON or OFF state. Alternatively, electronically programmable ON/OFF devices such as pass transistors can be used. Besides offering an easy mechanism for correcting in-service faults, they provide for yield enhancement reconfiguration with interconnection adaptation to a particular problem. However, they need more area since the state also has to be stored. The other advantage of using physically restructurable switches is that they typically introduce a smaller on-state resistance than electronic switches. Thus signal rise time degradation and propagation delay per switched link is considerably reduced. The switch programming is nonvolatile and need not be run each time on power-on. This is especially important for a large array with many PE's. The chief disadvantage is that they offer only a permanent solution and the connections cannot be reprogrammed at runtime to account for additional faults.

In systolic arrays, timing is important. It is desirable to minimize the additional delay caused by communicating over a logical link which could be made up of a series of physical links connected by switches. Based on an incremental distributed-RC model of electrical interconnection, the delay τ_{RC} introduced is roughly $N^2 R_{sw} C_{ln}$, where R_{sw} is the series resistance introduced by each switch and C_{ln} is the capacitance imposed by each link and N is the number of links in the path. For a typical minimum geometry pass transistor, this can be about 10 ns delay per cm of line length and per square of gate area. In our running example where one H line and one V line were used, it can be shown from Theorem 4 that there are at most three links per logical path. Assuming a link is 0.1 cm, this places an upper bound of 3 ns of the delay. Clearly it is preferable to keep the number of additional links per reconfigured path as small as possible. The following theorem establishes a bound on the maximum wire length and, therefore, bounds the maximum delay in signals. In practice, however, R and C need to be replaced by the actual number of H lines and V lines that were used in arriving at the solution.

Theorem 4: For a hexagonal array, with R spare rows and C spare columns, the length l, of any path p, between two logically adjacent cells satisfies the relation:

$$l \leq \max (2R + C + 1, 2C + R + 1).$$

Proof: a) Horizontal direction: From Theorem 1, it follows that the path p has two SE's on every H line between the cells; and one SE on every V line between. Since there can be at most R H lines and C V lines between the two, this means that path p can at most comprise $2R + C$ SE's. Hence, the maximum length in the horizontal direction, l_h is less than or equal to $2R + C + 1$.

b) Vertical direction: By similar arguments, it can be shown that the maximum length in the vertical direction, l_v is less than or equal to $2C + R + 1$.

c) Diagonal direction: From Theorem 1, we can deduce that this case is a combination of the above two. Every cell on path p which contains a *Straight* H line resembles a), and those that contain a *Straight* V line resemble b). Thus, the maximum length in the diagonal direction l_d, subject to postprocessing, is less than or equal to max $(2R + C + 1, 2C + R + 1)$. However, for type "c" cells, the path length increases by just 1 for each pair of an H line and V line. This has been found to be more the typical case. Hence, l_d is more often close to max $(R + 1, C + 1)$.

The theorem follows from the above three cases.

Q.E.D.

Postprocessing: A consequence of the automatic switch settings done by table lookup is that a reconfigured path may traverse at SE of type "c" twice. The presence of such a loop in the path is unnecessary and is eliminated in a postprocessing step, wherein the abovementioned SE's of type "c" are reconfigured appropriately. Note that these loops do not affect the correctness of the solution,

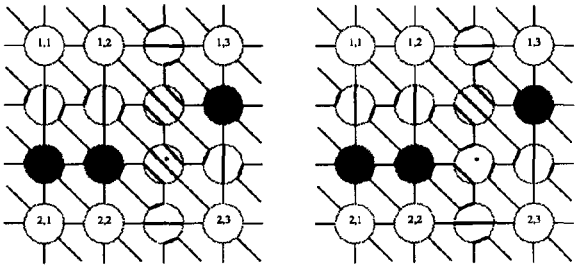


Fig. 11 Eliminating redundant links by postprocessing.

but their removal improves the performance by reducing the overall length. Fig. 11(a) shows how such a postprocessing procedure can detect double traversals of the reconfigured path between logical cells (1, 2) and (2, 3) at the cell marked with a * and reconfigure it as shown in Fig. 11(b). This step thus eliminates three additional links in the reconfigured path.

VIII. Conclusion

In this paper we have presented and analyzed a reconfiguration algorithm, HEX-REPAIR, intended for wafer-scale hexagonal processor arrays. Reconfiguration schemes can be evaluated by their fault-coverage characteristics and the accompanying switching overhead needed. The two are usually directly proportional to one another. For example, fault-stealing approaches have good fault coverage but need very large multiplexers and a large number of extra data links between processors. Consequently, they are not suitable for many hexagonal arrays used in digital signal and image processing applications, which often have relatively simple processors, each consisting of a few hundred transistors only.

HEX-REPAIR has been shown to be fairly robust even in the presence of multiple faults. Computational efficient techniques such as fault compaction and suitable heuristics, such as SE configuration by table-lookup, have been employed to get a solution whenever possible, in time which is polynomial in the number of faults. This is despite the fact that the original problem is NP-complete. The only extra hardware needed to implement this algorithm is a switch made up of 15 ON/OFF devices per processor. No extra data paths are introduced between processing elements. Also, the switch complexity is independent of the number of rows and columns of spare cells used. The correctness of the reconfigured solution and bounds on path length increase have been derived.

Acknowledgment

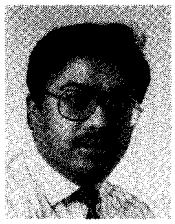
The authors would like to thank the Editor and other reviewers for their numerous comments which have considerably improved the content and presentation of the paper.

References

[1] S. Y. Kuo and W. K. Fuchs, "Spare allocation and reconfiguration in large area VLSI," in *Proc. Design Automation Conf.*, 1988, pp. 609–612.
[2] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice Hall, 1988.
[3] T. Noll, "Semi-systolic maximum rate transversal filters with programmable coefficients," *Systolic Arrays*, pp. 103–112, 1986.
[4] F. Distante, F. Lombardi, and D. Sciuto, "Array partitioning: A methodology for reconfigurability and reconfiguration problems," *Proc. ICCD*, 1988, pp. 564–567.
[5] R. Negrini, M. Sami, and R. Stefanelli, "Fault tolerance techniques for array structures used in supercomputing," *Computer*, pp. 78–87, 1986.
[6] M. Sami and R. Stefanelli, "Reconfigurable architectures for VLSI processing arrays," in *Proc. IEEE*, vol. 74, pp. 712–722, May 1986.
[7] F. Lombardi, M. G. Sami, and R. Stefanelli, "Reconfiguration of VLSI arrays by covering," *IEEE Trans. Computer-Aided Design*, vol 8, pp. 952–965, Sept. 1989.
[8] J. W. Greene and A. E. Gamal, "Configuration of VLSI arrays in the presence of faults," *J. Ass. Comput. Mach.*, vol. 31, pp. 694–717, Oct. 1984.
[9] T. Leighton and C. E. Leisorson, "Wafer-scale integration of systolic arrays," *IEEE Trans. Computers*, vol. C-34, pp. 448–461, 1985.
[10] S. K. Tewksbury, *Wafer Level Integrated Systems: Implementation Issues*. Kluwer Academic, 1989.
[11] D. Gordon, I. Koren, and G. M. Silberman, "Restructuring hexagonal arrays of processors in the presence of faults," *J. VLSI Comp. Syst.*, pp. 23–35, 1987
[12] F. Distante, M. G. Sami, and R. Stefanelli, "Reconfiguration techniques in the presence of faulty interconnections," in *Proc. Int. Conf. on Wafer Scale Integration*, 1989, pp. 379–388.
[13] I. Koren, "A reconfigurable and fault-tolerant VLSI multiprocessor array," in *Proc. 8th Ann. Symp. Comput. Architecture*, 1981, pp. 425–442.
[14] S. Y. Kuo and K. Wang, "Fault diagnosis in VLSI/WSI processor arrays," in *Proc. Int. Conf. on Wafer Scale Integration*, 1989, pp. 325–333.
[15] F. Lombardi, "Reconfiguration of hexagonal arrays by diagonal deletion," *Integration*, pp. 263–290, 1988.
[16] A. L. Rosenberg, "The Diogenes approach to testable fault-tolerant arrays of processors," *IEEE Trans. Computers*, vol. C-32, pp. 902–910, Oct. 1983.
[17] D. Sciuto, "Testability and design for testability for WSI systolic hexagonal arrays," in *IFIP Workshop on Wafer Scale Integration*, 1990, pp. 109–119.
[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*. New York: Cambridge Univ. Press, 1988, pp. 329–345.
[19] G. E. Farr and H. Schroder, "An evaluation of reconfiguration algorithms in fault tolerant process arrays," in *Proc. 5th MIT Conf. Advanced Research in VLSI*, pp. 131–148, 1988.

R. Venkateswaran (S'89) received the B.Tech. degree in computer science from the Indian Institute of Technology, Bombay, in 1988 and currently, he is working toward the Ph.D. degree at the University of Michigan.

In the summer of 1991, he worked at the Thomas J. Watson Research Center, Yorktown Heights, NY, on hierarchical compaction. His areas of interest include computer-aided design, with particular emphasis on wire layout. Other research interests include parallel architecture, fault tolerance and neural networks. He was the recipient of the IBM Fellowship for 1991–1992.



Pinaki Mazumder (S'84–M'87) received the B.S.E.E. degree from the Indian Institute of Science in 1976, the M.Sc. degree in computer science from the University of Alberta, Canada in 1985, and the Ph.D. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1987.

Presently he is an Associate Professor at the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. Prior to this, he was a Research Assistant at the Coördinated Science Laboratory, University of Illinois, and for over six years was with Bharat Electronics Ltd., India, (a collaborator of RCA) where he developed several types of analog and digital integrated circuits for consumer electronics products. During the summers of 1985 and 1986, he was a Member of the Technical Staff in the Naperville branch at AT&T Bell Laboratories. His research interest includes VLSI testing, physical design automation, ultrafast digital circuit design, and neural networks. He has published over 50 papers in IEEE and ACM archival journals and reviewed international conference proceedings. He is a Guest Editor of the IEEE DESIGN AND TEST magazine's special issue on memory testing, to be published in 1993.

Dr. Mazumder is a member of Sigma Xi, Phi Kappa Phi and ACM SIGDA. He is a recipient of Digital's Incentives for Excellence Award, National Science Foundation Research Initiation Award, and Bell Northern Research Laboratory Faculty Award.

Kang G. Shin (S'75–M'78–SM'83–F'92) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY in 1976 and 1978, respectively.

From 1978 to 1982 he was on the faculty of the Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the U.S. Air Force Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at University of California, Berkeley, and International Computer Science Institute, Berkeley, CA. He is Professor and Associate Chair of Electrical Engineering and Computer Science (EECS) for Computer Science and Engineering, the University of Michigan, Ann Arbor. He has authored/coauthored over 180 technical papers (about 80 of these in archival journals) and several book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, and robotics and automation. In 1987, he received the Outstanding IEEE TRANSACTIONS ON AUTOMATIC CONTROL Paper Award for a paper on robot trajectory planning. In 1989, he also received the Research Excellence Award from the University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called HARTS, to validate various architectures and analytic results in the area of distributed real-time computing.

Dr. Shin was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, the Guest Editor of the 1987 August special issue of IEEE TRANSACTIONS ON COMPUTERS on Real-Time Systems, and is a Program Co-Chair for the 1992 International Conference on Parallel Processing. He currently chairs the IEEE Technical Committee on Real-Time Systems, is a Distinguished Visitor of the Computer Society of the IEEE, an Editor of IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED COMPUTING, and an Area Editor of *International Journal of Time-Critical Computing Systems.*