Fault Tolerance Approaches For
3-Data Flow Systolic Arrays

Tarek Rizkallah Boulos

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

May 1991

ISBN 0-315-68784-3

ABSTRACT

Fault Tolerance Approaches For

3-Data Flow Systolic Arrays

Tarek Rizkallah Boulos

This thesis presents techniques to achieve fault tolerance for three directional data flow (3-data flow) systolic arrays both at run-time and at compile-time. Two run-time schemes are suggested. The first one is based on space-time mapping. This scheme, named 3DFT (three directional fault tolerant), is a fault tolerance approach based on fault masking, and it is applicable to a class of rectangular 3-data flow systolic arrays of latency greater than or equal to three. The main advantage of the scheme is its tolerance to the occurrence of multiple faults (permanent and transient). In addition, the overhead in hardware is minimal. A processing element (PE) supporting the concept is suggested. The fault assumption is that only one PE out of the three involved in doing the same computation can be faulty. The technique is not applicable in all cases; however, in the specific cases where it applies, fault tolerance can be achieved with a very low overhead. The technique can be easily extended to linear systolic arrays.

Another run-time fault tolerance scheme based on reconfiguration for unidirectional 3-data flow systolic arrays is also proposed. This distributed reconfiguration algorithm enables the cells of a processing array to dynamically restructure themselves based on local information. No a priori cell programming is required. The approach allows the reconfiguration of the three data flow paths (horizontal, vertical and diagonal). Transient faults can be tolerated provided there are sufficient spare processors to replace the faulty ones.

For compile-time fault tolerance, an extension of the RCS-cut approach (static reconfiguration) is presented to cover the case of unidirectional 3-data flow systolic arrays.

The RCS-cut approach may be used to reconfigure the array before applying any of the run-time techniques. The 3DFT approach can be applied to computational problems that map naturally on hexagonal arrays (matrix multiplication, transitive closure, etc.). The dynamic reconfiguration approach is more general and can be applicable at run-time on any computation that require 3-data flow systolic arrays.

ACKNOWLEDGMENTS

I would like to express my greatest gratitude to my thesis supervisor, Dr. R. Jayakumar for his guidance and encouragement throughout the course of this research. He made great contributions in its development.

I would also like to thank Dr. H. F. Li for his useful advice, criticism and for giving me the opportunity to do this work.

I am also grateful to Mr. Derek Pao for his very useful comments, criticism, helpful discussicns and proofreading that helped me on final completion of this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Systolic arrays [KuHT82] are most desirable for VLSI implementation because of their regularity and locality of data communications. As the integration scale increases dramatically, it is inevitable that there are faults (permanent or temporary) in the system. Hence fault-tolerant design is necessary. Yield and reliability are the two major concerns for fault-tolerance in VLSI. The fault-tolerance schemes should not only improve the yield, but also ensure correctness of the computation performed.

Faults in VLSI can be classified into two categories:

1. *Production faults*: faults introduced during the manufacturing process.
2. *Operational faults*: faults that arise during the operation of the circuit. These include aging, soft-error caused by alpha particles, coupling of signals, and hot electrons, etc. These can be further divided into two sub-classes, namely, *permanent faults and transient faults.*

One consequence of the scaling down of device dimensions is that VLSI circuits are more susceptible to operational faults. Hence, fault-tolerance schemes that can tolerate operational faults are desirable.

Fault-tolerance (FT) schemes can be classified as:

1. *Fabrication-time FT*: Yield is the major concern.
2. *Compile-time FT*: When faults occur after installation, the array is reconfigured to function properly, very often with degraded performance.
3. *Run-time FT*: In this case, the emphasis is on dependable computation. The effects of faults should be masked off or compensated with minimal time delay.

There are two fault-tolerance approaches proposed in the literature, namely, reconfiguration which is used for fabrication-time and compile-time FT, and fault-masking or concurrent error dctection and correction for run-time FT.

1.1 Run-time FT

1.1.1 Fault-masking

Fault masking stands for immediate recovery without noticing the occurrence of faults. In this approach, extra components are used to mask off the effect of a fault instantaneously. A common technique employed is the Tripie Modular Redundancy (TMR) [Ball69]. Three copies of the same computation are performed by three processors. Voting among the three will determine the correct result if only one of the three processors can fail. Direct implementation of this method often requires expensive duplications. Kim and Reddy [KimJ85] proposed a scheme that can reduce the number of processors required by a factor of two while tolerating only a limited class of fault distributions and presented the design of a linear array to perform matrix ($A = [a_{ij}]$) vector (X) multiplication. The vector X is duplicated over two consecutive time instances and the matrix A is triplicated spatially. Three copies of the computed result can be obtained at the same time by proper design of the interconnection links between the processing elements.

1.1.2 Concurrent Error Detection and Correction

Chan and Wey adapted the approach of recomputation with shifted operands in their design of fault-tolerant systolic arrays for band matrix multiplication [ChSW88]. Gulati and Reddy [Gula86], and Cosentino [Cose88] proposed another approach based on temporal redundancy. Gulati and Reddy presented a concurrent error detection (CED) scheme using 'Comparison with Concurrent Redundant Computation' (CCRC) which is based on a more appropriate cell level functional model, and its basic theme is to duplicate (and compare) the (sub) computations done in every cell of the array. The area overhead in CCRC is limited to one computational unit of the array and some additional error detection logic in every cell. Application of CCRC reduces the throughput to half, if the original implementation is capable of delivering one result every cycle. Cosentino presented a concurrent error correction scheme for systolic arrays in which the cells retain partial results rather than pass them to the neighbors.

Abraham et al [Huan84] proposed an algorithm based fault-tolerance scheme for matrix operations employing the row and column checksum technique. The algorithm is redesigned to operate on the encoded data and produces encoded output. The result vector also preserves the weighted checksum property which is used to correct error(s). Discrepancy in the checksums produced by the algorithm and calculated from the output data indicates an error. The located error can be easily corrected based on the checksum produced by the algorithm.

All of the above fault tolerance schemes are ad hoc designs. Recently, Li et al [LiHF89] characterized the CED capability of a systolic array in terms of the latency of computation which is determined by the space-time mapping. Latency is the time delay before a processor performs a computation after it has just completed one. They also proposed a way of achieving CED in systolic arrays by incorporating redundancy systematically.

1.1.3 Transient Faults

It is worth mentioning that transient faults are tolerated mostly at run-time. It is reported in [Siew82] that the occurrence of transient faults, mainly due to temporary environmental changes, is ten times more frequent than that of permanent faults. In addition, with the present possibility of reduced voltage levels for VLSI devices and the subsequent reduction in noise margins [Barb81], susceptibility of dense VLSI circuits to transient faults also increases.

1.2 Reconfiguration

A common approach for fabrication-time and compile-time fault tolerance is reconfiguration. Reconfiguration is equivalent to finding an embedding of the logical array onto the physical array such that faulty PEs do not participate in the computation. Several reconfiguration algorithms have been proposed. Leighton and Leiserson [Leig85] proposed a reconfiguration algorithm for wafer scale integration of systolic arrays based on divide and conquer method. The interconnection wires are programmed by laser beams. In Rosenberg's method [Rose83], the processing cells are fabricated with uncommitted interconnection wires and controllable switches. Processing cells which are non-faulty are connected together to obtain the desired array.

Kung and Lam [KuHT84] proposed a systolic fault-tolerance scheme (RCS-cut) which maintains the original data flow pattern and wire length. In this scheme, data move through all the cells. At faulty cells, data items are simply delayed with bypass registers for one cycle and no computation is performed. The processor array is modelled as a directed graph, with the nodes denoting the processing cells and the edges denoting the communication links. A cut is defined as a set of edges that "partition" the nodes into two disjoint sets, namely, the source and the destination sets, with the property that these edges are the only ones between these sets, and are directed from the source to the destination set. Two designs are equivalent if given an initial state of one design, there exists for the other design an initial state such that (with the same input from the host) the two designs produce the same output values (although possibly with different delays). The restructuring of the faulty array is carried out by choosing a sequence of cuts whose edges cover all the faulty cells in the array.

Koren [Kore81] presented a distributed reconfiguration algorithm to restructure two-dimensional processor arrays. Restructuring is done by distributing the following type of structuring messages:

M (structure code, level within the structure, direction).

The header of the message M is the code of the desired structure, such as LA for linear array. The level number indicates the position of the processor receiving the message M in the array. The direction *d* (*d* is an element of {N, E, S, W}) indicates the neighbor to which the next structuring message should be transmitted. A processing cell receiving a structuring message will transmit either another structuring message or an acknowledgment (a positive acknowledgment if the construction is completed or a negative acknowledgment if it is impossible to complete the construction) to its neighbor(s). As the structuring messages percolate through the physical array, the corresponding data switches are programmed accordingly. To avoid using a faulty cell, the cells in the row and column containing faulty cell(s) are programmed to function as connecting cells. In case of unsuccessful reconfiguration (receiving a negative acknowledge), backtracking is required. There are two drawbacks to this method.

1. Even though the reconfiguration algorithm is distributed, setting of the data switches cannot be changed without interrupting the computation.
2. Utilization of non-faulty cells is low because each faulty cell may cause the removal of one whole row and one whole column of cells.

All of the above mentioned reconfiguration schemes are "static" in the sense that the interconnection links are externally enforced (laser programmed or controllable data switches). Once the array is reconfigured, the setting of the interconnection links cannot be changed without interrupting the computation. Hence, they cannot tolerate transient faults. A review and more details concerning the previous two approaches will be given in chapter 3.

A special type of the 3-data flow systolic arrays is the hexagonal array where the processors communicate with each other through a hexagonal array network [KuHT79, Rote86]. The hexagonal array structure enjoys the property of symmetry in three directions. An example of algorithms that use the hexagonal array as the communication geometry is the matrix multiplication problem [Mead80]. It is worth mentioning that algorithms employing multi-directional data flow can realize complex computations without violating the simplicity and regularity constraints. Moreover, these algorithms do not require separate data loading or results unloading phases. Notice that hexagonal connection supports data flows in more directions than square connections and the two structures are of about the same complexity as far as implementation is concerned.

Gordon et al [Gord87] extended the concept described above [Kore81] in order to achieve fault tolerance in hexagonal arrays. Kumar [Kuma91] proposed two reconfiguration designs for hexagonal systolic arrays, one for compile-time and the other for fabrication-time.

Pao [PaoD87, LiHF87] presented a run-time fault tolerance scheme based on reconfiguration, for uni-directional 2-data flow systolic arrays. In this approach, the data paths are set up dynamically during the computation and faulty PEs are bypassed systolically.

This thesis presents techniques to achieve fault tolerance for 3-data flow systolic arrays both at run-time and at compile-time. Two run-time schemes are suggested. The first one is based on space-time mapping. This scheme named 3DFT (three directional fault tolerant), is a fault tolerance approach based on fault masking and it is applicable to a class of rectangular three directional data flow systolic arrays of latency greater than or equal to three. Informally, latency characterizes the rate at which data of a problem instance can be bumped into the systolic array. The main advantage of the scheme is its tolerance to the occurrence of multiple faults (permanent and transient). In addition, the overhead in hardware is minimal. A processing element (PE) supporting the concept is suggested. The fault assumption is that only one PE out of the three involved in doing the same computation can be faulty. The technique is not applicable in all cases; however, in the specific cases where it applies, fault tolerance can be achieved with a low overhead. The technique can be easily extended to linear systolic arrays.

Another run-time fault tolerance scheme based on reconfiguration for unidirectional 3-data flow systolic arrays is also proposed. This distributed reconfiguration algorithm enables the cells of a processing array to dynamically restructure themselves based on local information. No a priori cell programming is required. Transient faults can be tolerated provided there are sufficient spare processors to replace the faulty ones.

For compi!e-time fault tolerance, an extension of the RCS-cut approach [KuHT84, LamC89] (static reconfiguration) is presented to cover the case of unidirectional 3-data flow systolic arrays.

Chapter 2 reviews the space-time approach and the conditions to get a systolic mapping. The notions of latency, concurrent redundant computation (CRC) and concurrent error detection (CED) are presented. The conditions of applicability of the 3DFT approach are explained. A processing element (PE) supporting the design is suggested. The effects of different mappings are discussed. The fault model and the types of tolerated faults is covered. The advantages of using the 3DFT, followed by a comparison with Kim and Reddy's technique [KimJ87a] for getting a fault-tolerant systolic array to the LU decomposition problem is presented. Also, a comparison with Cosentino's approach is discussed. Chapter 3 discusses the extension of the application of the RCS cut approach to unidirectional 3-data flow systolic arrays. It also includes a comparison of restructuring techniques for hexagonal arrays, like Gordon's approach [Gord87], Kumar's technique [Kuma91] and the RCS-cut approach. Chapter 4 presents a dynamic, distributed algorithm for unidirectional 3-data flow systolic arrays which restructure the processing array based on local information only. Chapter 5 is the summary of the thesis.

Chapter 2

RUN-TIME FAULT TOLERANCE FOR THREE DIRECTIONAL DATA FLOW SYSTOLIC ARRAYS

2.1 Mapping Data-flow Computation Into Systolic Arrays

Systolic arrays have been used to implement iterative data flow computations successfully. A three-dimensional data flow computation involves a set of computation sites $\{C_D = (i,j,k)\}$ such that at each site $s = (i,j,k) \in C_D$ a set of computations $f_1(i,j,k)....f_r(i,j,k)$ are performed. In order for a computation structure to be implementable in a systolic array, the conceptual computation sites $\{C_D = (i,j,k)\}$ must be mapped into $C_S= \{(t,x,y)\} \subset Z^3$ (where Z denotes the 1-dimensional integer space, t denotes time and (x,y) represents the two-dimensional space normally available for VLSI systolic array embedding). The general idea of space-time mapping is as follows:

1. The algorithm is formulated as a system of linear uniform recurrence equations. If a variable propagates without being modified, it is called a *used variable*, otherwise it is called a *generated variable.*
2. Data dependency vectors derived from the loop indices are organized as a dependency matrix $D = \{d1, d2,...,dr\}$.
3. A linear transformation T is then applied to the dependency matrix D to obtain a new dependency matrix (systolic matrix) $\Delta = TD$ whose first row is strictly negative while the elements in the other rows are one of $\{0,1,-1\}$ in case of a universal systolic array.

When D is mapped into a systolic array using a one-one transformation, $T = [t_{ij}]$, it yields $S = (\Delta, C_S)$, written $\Delta = TD$, where $C_S = \{(t,x,y)\}$ is spanned by

$$\begin{bmatrix} t \\ x \\ y \end{bmatrix} = T \begin{bmatrix} i \\ j \\ k \end{bmatrix} .$$

Systematic procedures to derive valid transformations have been proposed in [Mold83; Fort85; LiGJ85], which involve an exhaustive search. Zhang and Li [LiHF89b]studied the problem of mapping a general iterative algorithm with non-unity increment/decrement steps onto a systolic array using a space-time transformation. They have developed the following necessary and sufficient conditions for the existence of such a space-time mapping:

Transformation $T = [t_{ij}]$ correctly transforms D into a systolic computation S [LiHF89b] if and only if

1. $|T| \neq 0$,
2. $\delta_{1j} < 0$ for all j, $1 \leq j \leq r$ (where "r" number of dependency vectors), and
3. lcm $(p_{11}, p_{21}, p_{31}) | \Delta i$; lcm $(p_{12}, p_{22}, p_{32}) | \Delta j$; lcm $(p_{13}, p_{23}, p_{33}) | \Delta k$ ("lcm" denotes " least common multiple", "|" denotes "divides relation and Δi, Δj and Δk are the step sizes) where $t_{ij} = q_{ij}/ p_{ij}$, and q_{ij}, p_{ij} are relatively prime.

They also presented a heuristic procedure to determine T.

It is possible to verify if there is a transformation matrix $T = [t_{ij}]$ that correctly transforms D into a specific systolic computation Δ in cases where D is a non-singular square matrix of integers since $T = \Delta D^{-1}$, where D^{-1} is the inverse of D and T should satisfy the conditions stated above. It is worth mentioning that in this case the transformation matrix T is unique.

2.2 Latency

Latency is one of the most important parameters in embedding a data flow computation into a systolic array. Informally, latency characterizes the rate at which data of a problem instance can be bumped into the systolic array. Consider the case of mapping a 3-dimensional iterative computation $D = (D, C_D)$ with $C_D = \{(i,j,k)\}$ into a systolic array S

= (Δ, C_S) with $C_S = \{(t,x,y)\}$. Conceptually, latency is the time delay Δt_{min} before a computational site (x,y) will be used again to perform another operation after it has just completed one. We know that a computation in $\{(i,j,k)\}$ is mapped into corresponding sites in $\{(t,x,y)\}$. In [LiHF89b] the latency of a systolic computation Δt_{min} was characterized in terms of the linear transformation T and the increment/decrement step sizes Δi, Δj, Δk as

$$\Delta t_{min} = \left| \frac{|T| \Delta i \Delta j \Delta k}{gcd(|T11| \Delta j \Delta k, |T12| \Delta i \Delta k, |T13| \Delta i \Delta j)} \right| \qquad (1)$$

where $|Tij|$ is the (i,j)-th co-factor of T.

Using the concept of latency, Li et al developed the following theory for concurrent error detection (CED) in systolic arrays using concurrent redundant computation (CRC).

2.3 Concurrent Redundant Computation (CRC) in Systolic Arrays

Suppose (D, C'_D) is a redundant version of (D, C_D). Consider

(D, C_D) ------T-----> (Δ, C_S) where $C_S = \{(t, x, y)\}$ and

(D, C'_D) ----T-----> (Δ, C'_S) where $C'_S = \{(t + dt, x + dx, y + dy)\}$.

We have

$$\begin{bmatrix} dt \\ dx \\ dy \end{bmatrix} = T \begin{bmatrix} di \\ dj \\ dk \end{bmatrix}. \qquad (2)$$

The computations (Δ, C'_S) and (Δ, C''_S) are *concurrent redundant computations* (CRC) of (Δ, C_S) in a systolic array implementation if and only if

- $C'_S \subset Z^3$ and $C''_S \subset Z^3$,
- $C'_S \cap C''_S \cap C_S = \phi$, and
- dt, dx and dy are constants.

Note that in general

- $dt \neq 0$: original computation and the corresponding redundant computation are performed at different time,

- dx = 0 and dy = 0: original computation and the corresponding redundant computation are performed at the same PE,

- dx $\neq$ 0 (dy $\neq$ 0): the distance between original computation and the corresponding redundant computation in x (y) direction is dx (dy)- additional rows of PE's (columns of PE's).

Concurrent error detection (CED) using concurrent redundant computation (CRC) was first proposed by Gulati and Reddy [Gula86]. This approach is algorithm independent and can be applied to a class of systolic arrays. Wu [WuCC87] presented a similar approach which is applicable to unidirectional data flow linear systolic arrays. Cosentino [Cose88] proposed a concurrent error correction scheme which is based on the CRC approach. The latter scheme is restricted to a class of systolic arrays whose latency is 2 in which the generated variables must stay in the cell.

2.4 Conditions of applicability of the 3-Directional data flow fault tolerance approach (3DFT)

Our technique (3DFT) is based on space-time mapping to achieve run-time fault tolerance for three directional data flow systolic arrays. Many computational problems fall into this category and map naturally on hexagonal systolic arrays [KuHT79,Mead80], e.g. matrix multiplication, LU decomposition, transitive closure [Rote85], etc. The scheme is based on fault masking and hardware redundancy under the assumption that the latency of computation is equal to or greater than three. The main advantage of the scheme is its tolerance to the occurrence of multiple faults (permanent and transient). The 3DFT technique is applicable to a class of rectangular three-data flow systolic arrays whose latency is equal to or greater than 3 (every processor will be idle for at least two cycles) and where the generated variable is non-stationary. Two redundant computations can be done concurrently with the original one. The fault assumption is that only one of three computation sites can be faulty.

2.4.1 Min-Overhead CRC

If $\Delta t_{min} \geq 3$, then there are two CRCs (Δ, C'_S) and (Δ,C''_S) of (Δ, C_S), both with (dt = 0, dx = ± 1, dy = 0) or with (dt = 0, dy = ± 1, dx = 0). As from the definition of latency above, every processor on the array will be idle for at least two cycles. The two redundant computations will take place during these two cycles.

This is equivalent to doing the three computations concurrently in three neighbor processors. The execution time will be the same as that of the original algorithm plus two extra time units due to the increase in array size. The array size will increase by only two extra rows (columns) of PEs. Hence this configuration, which results in having the minimum increase in the number of processors, is named *Min-Overhead CRC*. Notice also that this choice will result in minimum communication between the processors.

Example: Matrix multiplication

```
FOR i:= 0 TO N STEP 1 DO
    FOR j:= 0 TO N STEP 1 DO
        FOR k:= 0 TO N STEP 1 DO
            BEGIN
              A(i,j,k) := A(i,j-1,k);
              B(i,j,k) := B(i-1,j,k);
              C(i,j,k) := C(i,j,k-1) + A(i,j,k) * B(i,j,k)
            END
```

For this computation, A and B are the used variables, while C is the generated variable.

$$D = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \qquad T = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix} \qquad \Delta = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 1 & 0 \\ 1 & 1 & -1 \end{bmatrix}.$$

Using this transformation matrix T and $\Delta i = \Delta j = \Delta k = 1$, the latency Δt_{min} can be calculated using (1) to be $\Delta t_{min} = 4$.

It is advantageous to solve this problem using a hexagonal array rather than a rectangular array [KimJ87a]. Table 1 gives a summary of the comparison. The size of the hexagonal array depends on the bandwidth of the matrices A and B.

Table 1. Comparison of PE complexity and time taken to solve matrix multiplication problem (C = A x B)

		A = Full Matrix	A = Band Matrix	A = Band Matrix
		B = Full Matrix	B = Full Matrix	B = Band Matrix
Rectangular	#PE (A)	N x N	N x N	N x N
Array	Time (T)	O(4N)	O(4N)	O(4N)
	A x T	$O(4N^3)$	$O(4N^3)$	$O(4N^3)$
Hexagonal	#PE (A)	(2N-1) x (2N-1)	P x (2N-1)	P x Q
Array	Time (T)	O(3N)	O(2N)	O(N)
	A x T	$O(12N^3)$	$O(4PN^2)$	O(QPN)

A and B are NxN matrices.

For band matrix A (B), assume that bandwidth is P (Q) << N.

In our example, let us assume that both A and B are full matrices of size 3. This results in a hexagonal array of size 5x5. Fig 1.a shows the original data flow in the hexagonal array. Note that in any row or column of the network, only one out of three consecutive processors is active at any given time.

Choosing the Min-Overhead CRC dt = 0, dx = 1, dy = 0 (di = -0.5, dj = 0.5, dk = 0) the first redundant computation is as in Fig 1.b; and with dt = 0, dx = -1,dy = 0 (di = 0.5, dj = - 0.5, dk = 0) the second redundant computation is as in Fig 1.c. Since $C'_S \cap C''_S \cap C_S = \phi$ and the three input sequences do not overlap, a fault tolerant systolic array can be constructed by merging the three computation sites. Two snapshots (at time =2 and time =3) showing the computations performed by the processor is presented in Fig 2.a and Fig 2.b, respectively. The new hexagonal array size is 7x 5.

Note that a hexagonal array can be remapped to a rectangular array with diagonal input and output. For the hexagonal array in Fig 1.a, the equivalent rectangular array is shown in Fig 3.

For $T = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}$ $\Delta = \begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & 1 \\ -1 & 0 & 1 \end{bmatrix}$, and $\Delta t_{min} = 3$.

We can choose the Min-Overhead CRC such that dt = 0, dx = 1,dy = 0 (di = 2/3, dj = -1/3, dk = -1/3) for the first redundant computation and dt = 0, dx = -1,dy = 0 (di = -2/3, dj = 1/ 3, dk = 1/3) for the second.

2.4.2 Different Mappings

It is easy to see that with the Min-Overhead CRC a cluster of radius one cannot be tolerated since that will result in having two or three of the PEs involved in doing the same computation being in the faulty cluster. Assume that the center of the cluster is the PE at (x,y), then for a cluster of radius 1 the following PEs are assumed faulty:

(x-1,y - 1), (x-1,y), (x,y-1), (x,y+1), (x+1,y) and (x+1,y+1)

as shown in Fig. 4.

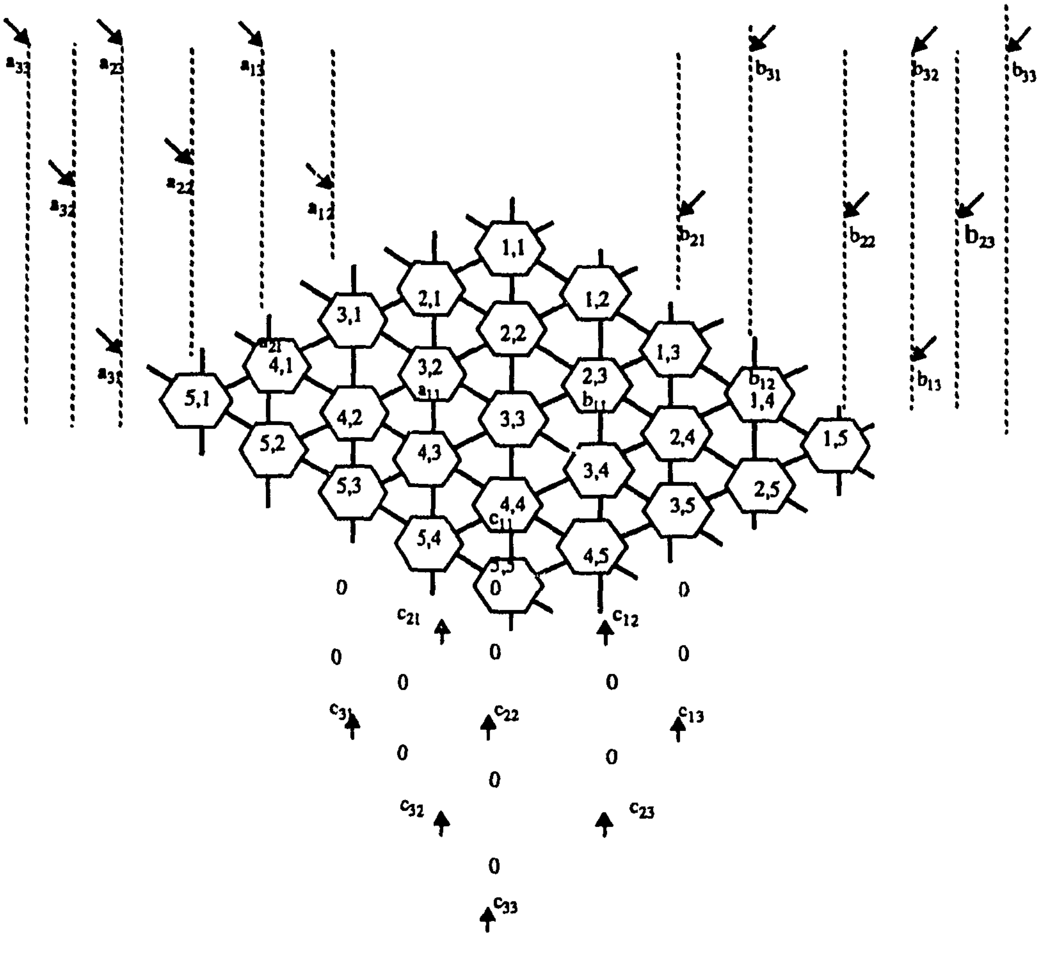


Fig 1.a Systolic Array for matrix multiplication (C = A x B)

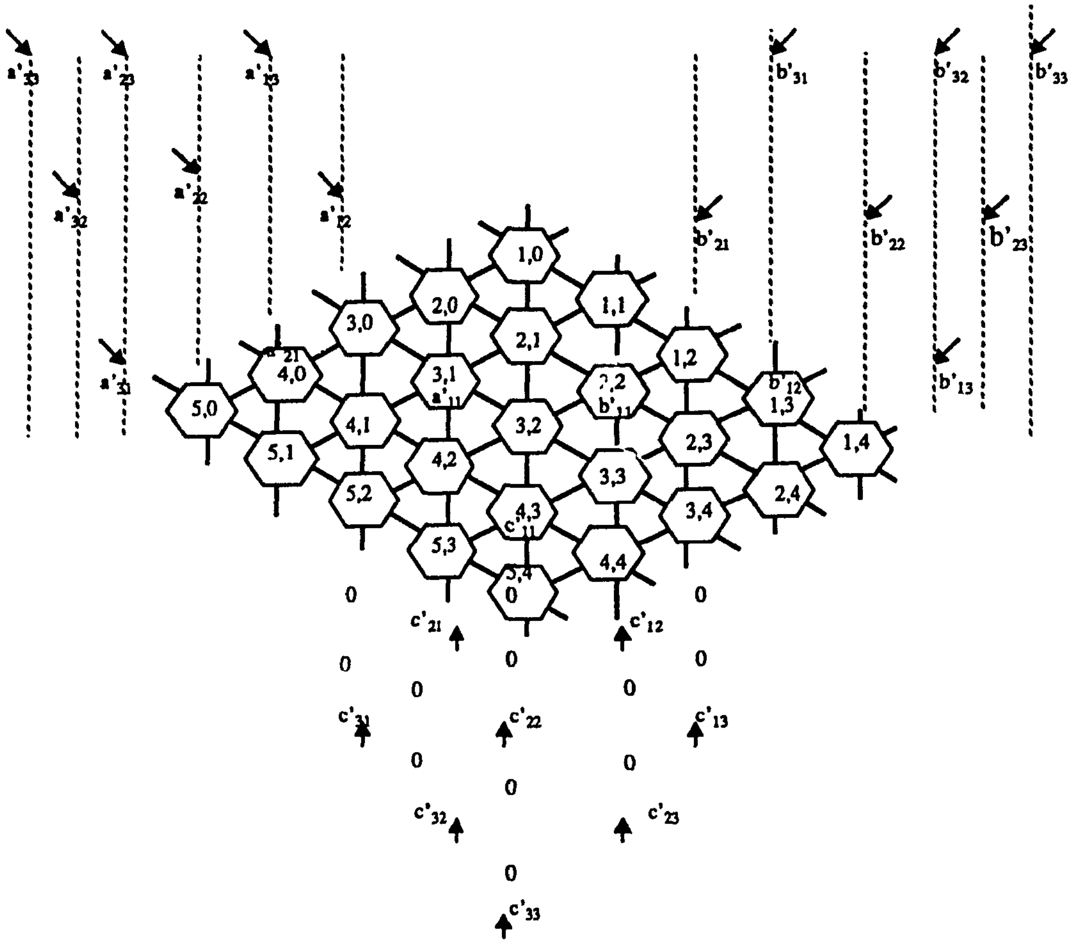


Fig 1.b CRC Systolic array of a redundant computation (S')

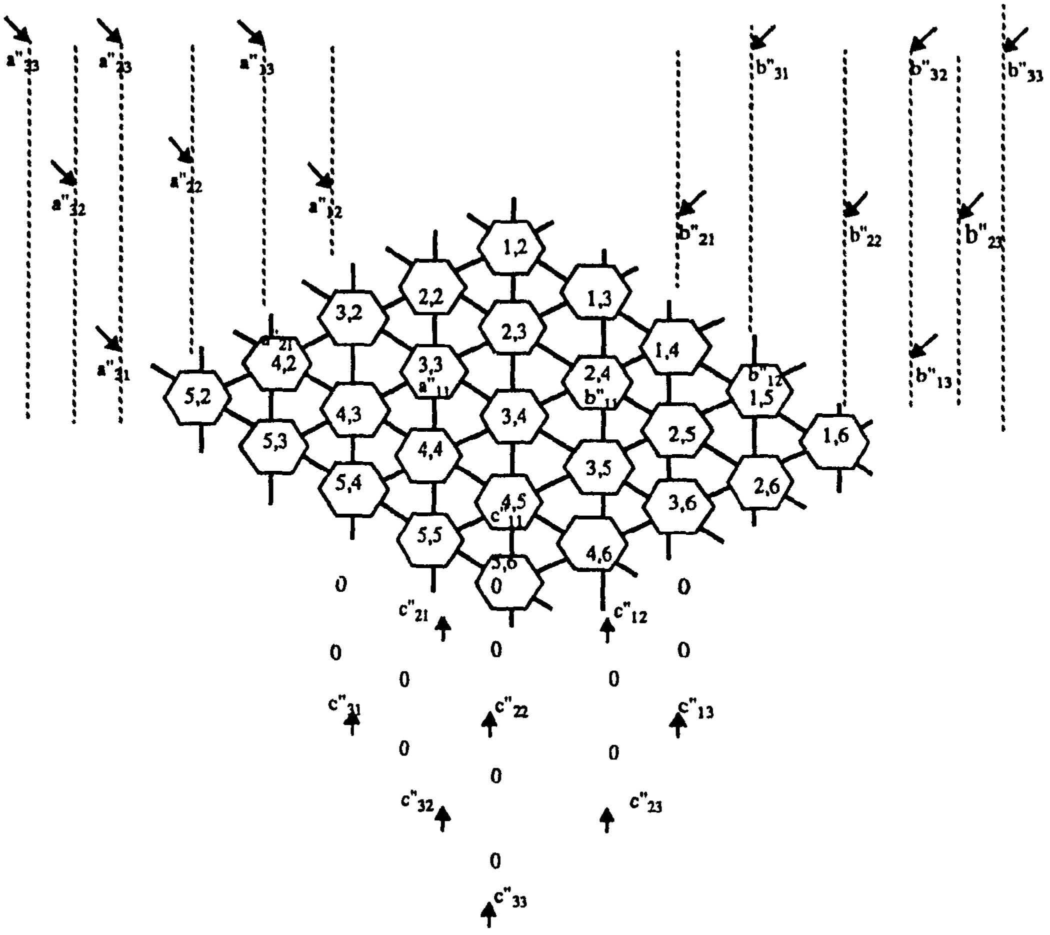


Fig 1.c Systolic array of another redundant computation (S")

time = 2

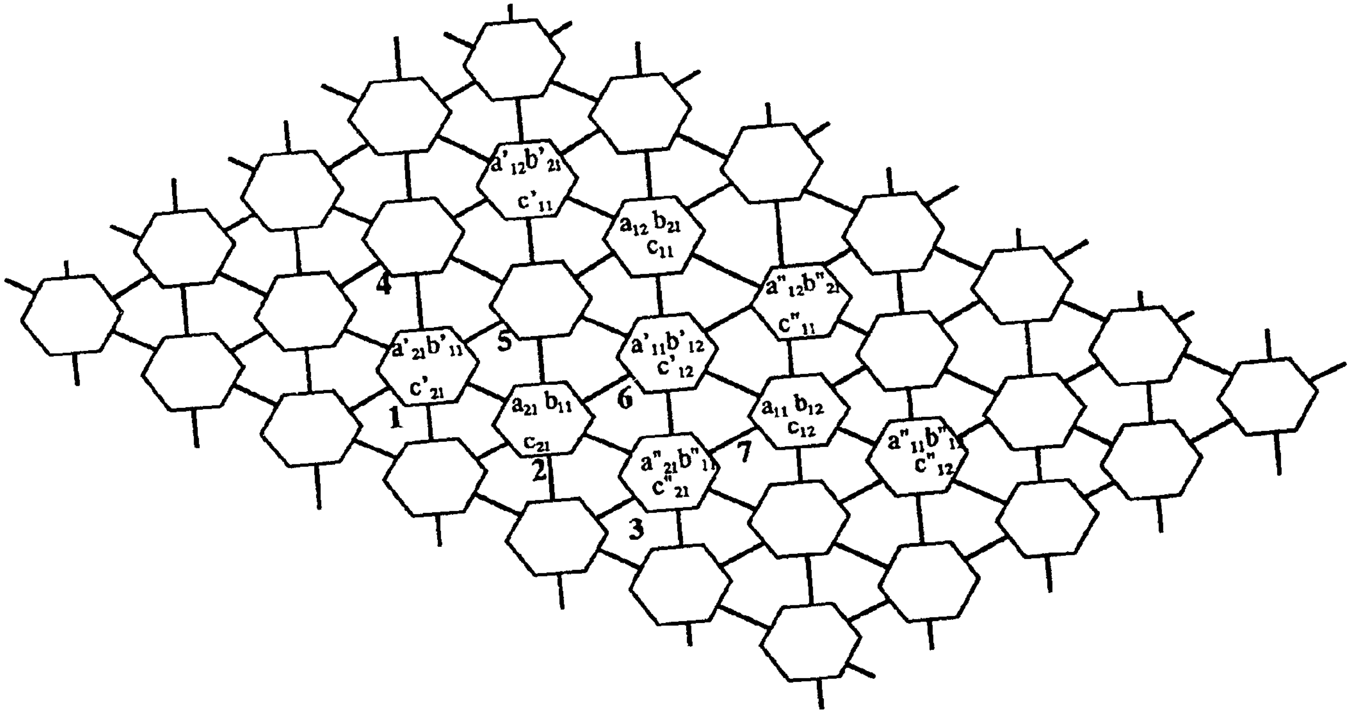


Fig 2.a At time =2 , the operation on every PE during the 3 x 3 matrix multiplication problem.

time = 3

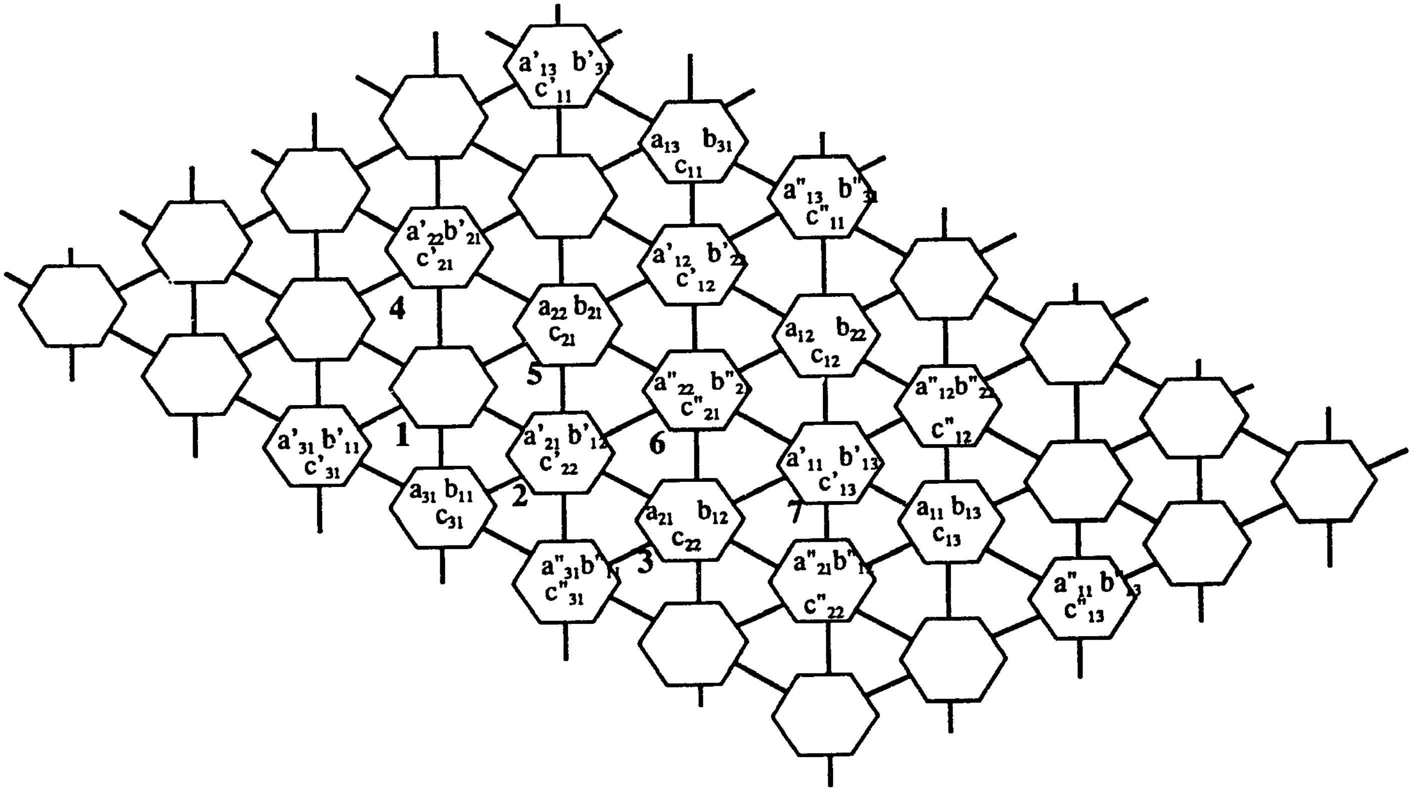

a'_{13} b'_{31} c'_{11}
a_{13} b_{31} c_{11}
a''_{13} b''_{31} c''_{11}
$a'_{22}b'_{21}$ c'_{21}
a'_{12} b'_{22} c'_{12}
4
a_{22} b_{21} c_{21}
a_{12} b_{22} c_{12}
5
a''_{22} b''_{21} c''_{21}
$a''_{12}b''_{22}$ c''_{12}
a'_{31} b'_{11} c'_{31}
1
a'_{21} b'_{12} c'_{22}
6
a'_{11} b'_{13} c'_{13}
a_{31} b_{11} c_{31}
2
a_{21} b_{12} c_{22}
7
a_{11} b_{13} c_{13}
$a''_{31}b''_{11}$ c''_{31}
3
$a''_{21}b''_{12}$ c''_{22}
a''_{11} b''_{13} c''_{13}

Fig 2.b Active PEs at time = 3.

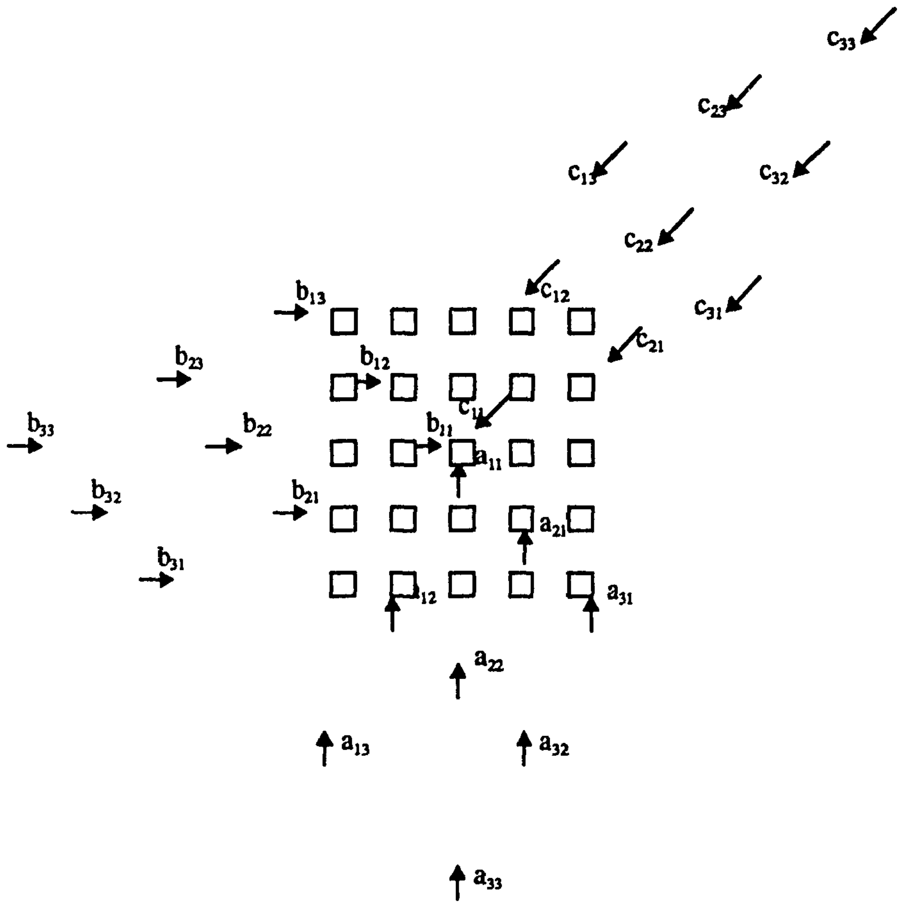


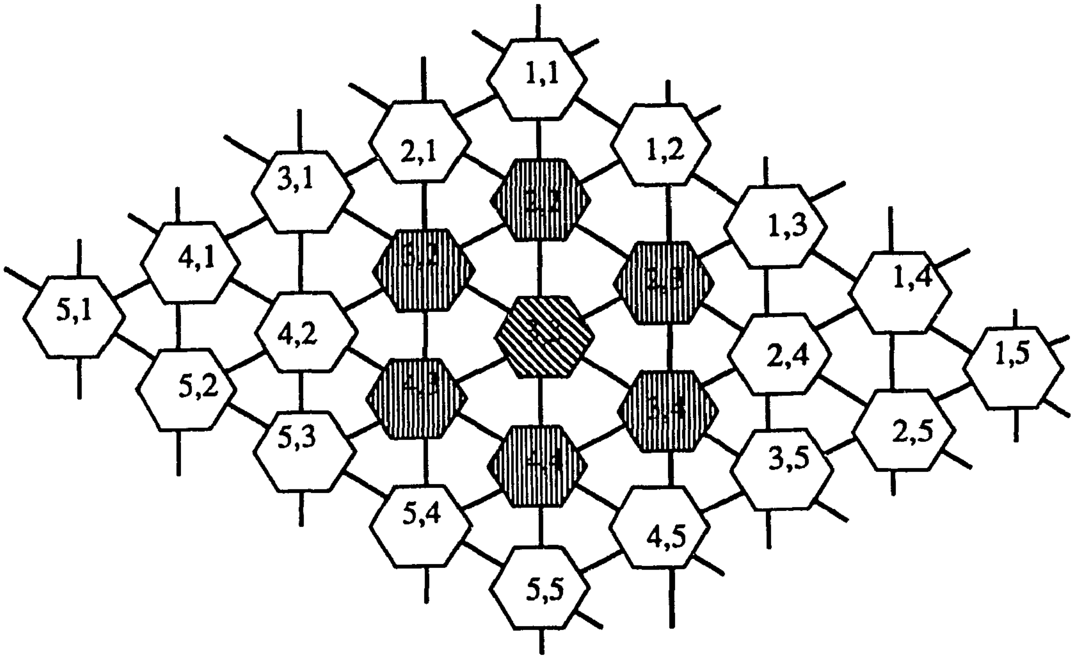
Fig 3. Rectangular array of size 5 x 5 with diagonal input and diagonal output showing the data flow for the matrix multiplication problem (N = 3).

In order to choose a mapping other than the Min-Overhead CRC, equation 2 in section 2.3 has to be solved for the values of (dt, dx, dy) such that the conditions of CRC in section 2.2 are satisfied for the two redundant computations. To have the three computations done concurrently, dt should be equal to zero as discussed in section 2.3. The effects of choosing different mappings are:

- longer communication lines,
- array size will increase drastically,
- types of tolerated faults will change depending on the locations of the redundant computation sites, and
- if either $|dx| \geq R + 1$ or $|dy| \geq R + 1$, then a single fault cluster of radius R may be tolerated at run time.

EXAMPLE

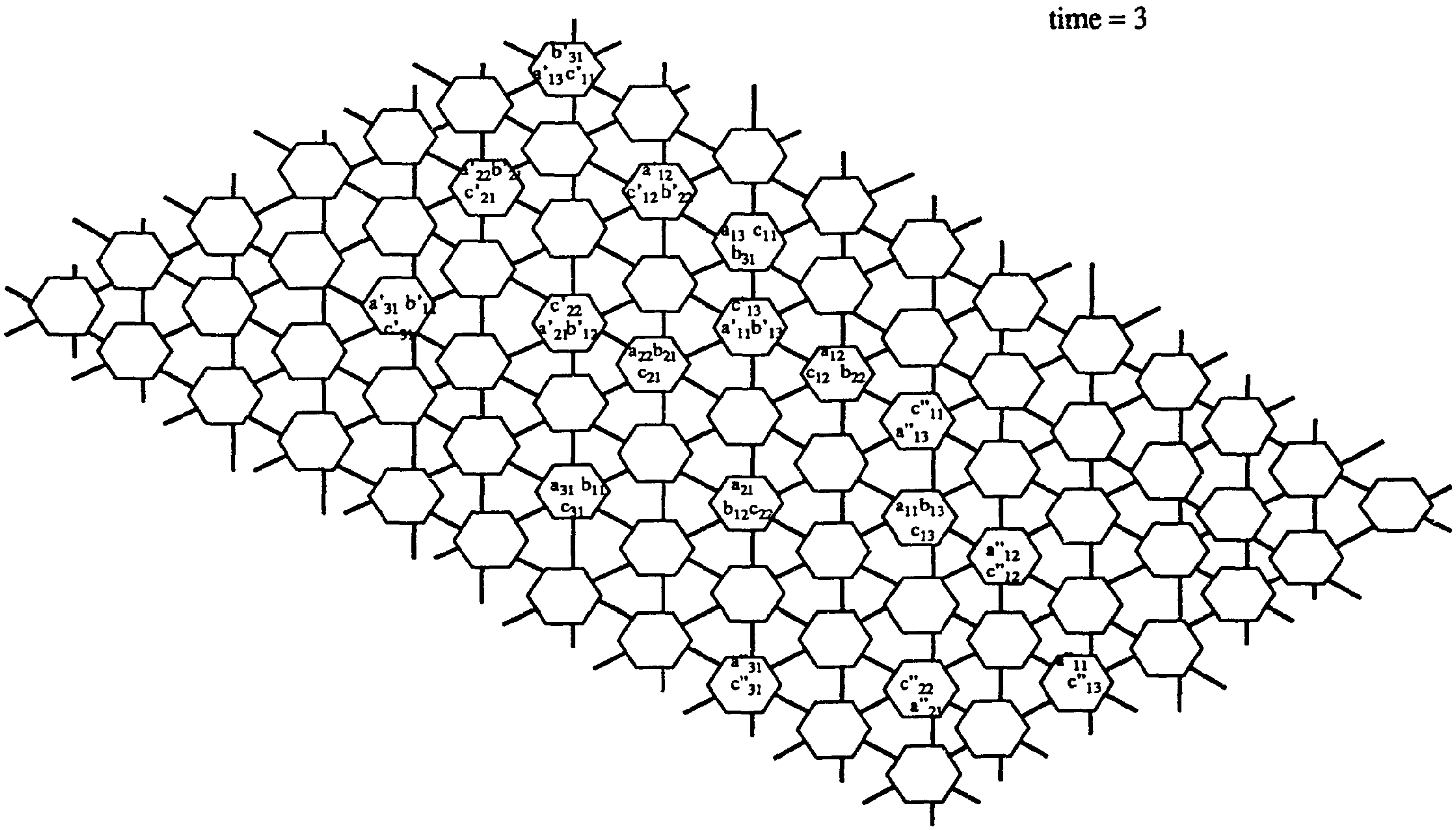
In the matrix multiplication problem given previously, other possible redundant computation sites that satisfy the conditions described in section 2.3 can be found by choosing ($dt = 0$, $dx = \pm 2$, $dy = \pm 1$). Note that this will result in having an array of size 11x7. Fig. 5 shows a snapshot (at time = 3) of the computation that can tolerate a cluster of radius 1 with this mapping.



▨ PE (x,y) Center of the Cluster

▥ PE affected on radius one of PE(x,y)

Fig. 4 Cluster of radius one.



time = 3

b'_{31} a'_{13} c'_{11}
a'_{22} b'_{21} c'_{21}
a'_{12} c'_{12} b'_{22}
a_{13} c_{11} b_{31}
a'_{31} b'_{11} c'_{31}
c'_{22} a'_{21} b'_{12}
c'_{13} a'_{11} b'_{13}
a_{22} b_{21} c_{21}
a_{12} c_{12} b_{22}
c''_{11} a''_{13}
a_{31} b_{11} c_{31}
a_{21} b_{12} c_{22}
a_{11} b_{13} c_{13}
a''_{12} c''_{12}
a''_{31} c''_{31}
c''_{22} a''_{21}
a''_{11} c''_{13}

Fig 5. The systolic array for the matrix multiplication problem with the new mapping to overcome a fault cluster of radius one.

2.5 PE Design and Functionality

Each processing element (PE) can be considered to have two parts: a *Computation Unit* (CU) that includes both the necessary elements to do the computation and the latches, and a *Voting Unit* (VU) which includes three identical voters, a decoder and an OR gate. Fig.6 shows a block diagram of the processing element.

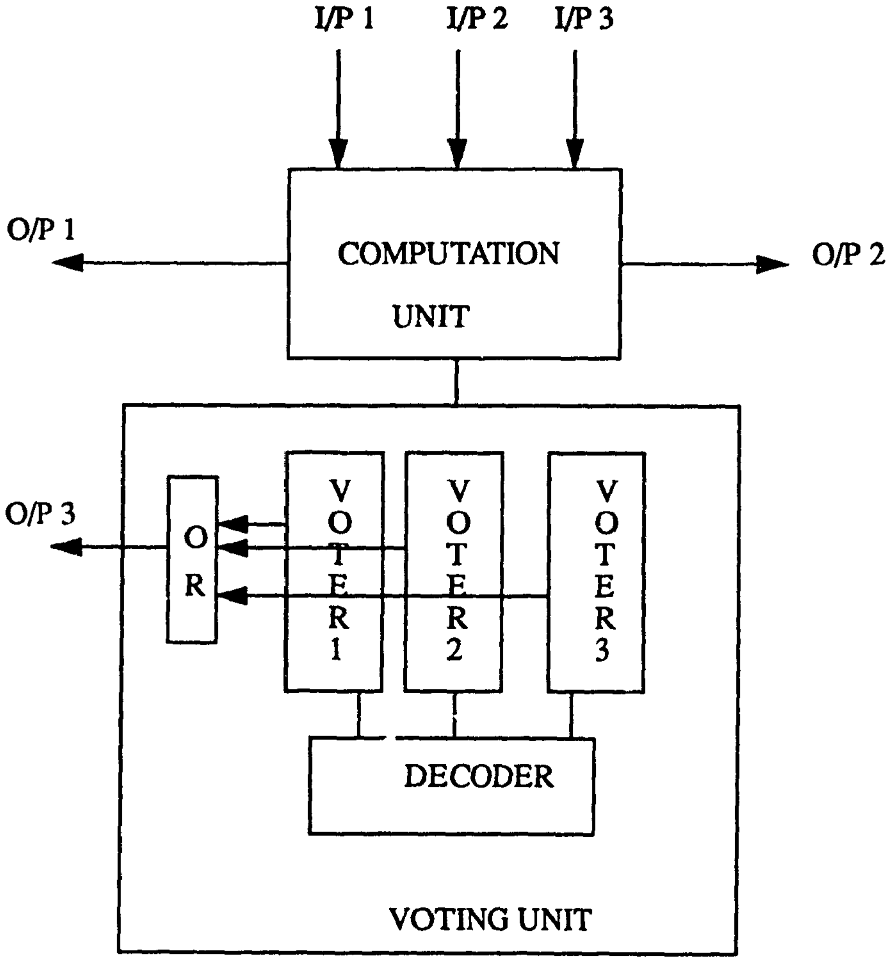


Fig. 6 Processing element block diagram (three inputs, three outputs, one generated variable).

Only one of the three voters is active during a clock cycle depending on the type of computation. There are 3 types of computations, the original and the two redundant ones. A binary code can be associated with each type of computation, for example, 10 for the original computation, 01 for the first redundant computation and 11 for the second redundant computation. This code is a control signal for the (2 x 4) decoder and can be sent with any of the inputs or outputs and enables one of the voters. Every voter gets one of the inputs from the computation unit output, which is denoted as c (the output of the adder in the case of the matrix multiplication example). As for the other two inputs, it will get them from the output of the next or previous computation units depending on the type of computation occurring in the PE and also on the locations of the computation sites. An OR gate collects the output of the three voters and outputs the final value which will propagate to the next stage to continue the computation.

The overhead in the necessary hardware is the Voting Unit (VU) which includes three identical voters, the 2 x 4 decoder and an OR gate. All of these elements are simple combinational circuits which are easy to design. Keeping in mind that only one of the three voters will be active at any time, then the delay of the voting unit is the sum of the propagation delays through the decoder, a single voter and an OR gate.

2.5.1 Example

The design discussed below is for the Min-Overhead CRC case (dt = 0, dx = ± 1, dy = 0). A deviation from this configuration will require the adjustment of the inputs to every voter.

Going back to the matrix multiplication problem, the computation unit includes a multiplier, an adder and three latches. For the voting unit the other two inputs to the voters are denoted as c(PE-1), c(PE+1), c(PE-2) and c(PE+2) in case of (dt = 0, dx = ± 1, dy = 0). Fig. 7 shows the design of the processing element for this problem.

To show which voter on the different PEs gets activated, the snapshots presented earlier will be used. At a certain clock cycle the following scenario will take place. In Fig 2.a PE 1 does the first redundant computation. The decoder receives control signal 01 and enables voter 1 of PE 1. Voters 2 and 3 are both not active. Simultaneously, PE 2 performs the original computation and has only voter 2 active and PE 3 does the second redundant computation and has only voter 3 active. During the next clock cycle (Fig 2.b), PE 1 is not used at all, PE 2 performs the first redundant computation and has voter 1 active and PE3 does the original computation and has voter 2 active.

NOTE: The lines that carries the code identifying the type of computation are not shown in the figures for simplicity.

2.6 Tolerated faults in 3DFT

An appropriate way to deal with failures in VLSI circuits is at the functional level. Therefore, we assume that the effects of faults will be seen only at small parts of a large network in the form of altered values of the outputs. The functional level chosen is a cell of the array and is thought to include the output links emanating from the cell. No fault-free assumption of any of the elements of either the CU or the VU is made. The following discussion considers the case of Min-Overhead CRC (dt = 0, dx = ± 1, dy = 0), that is, CRC is done on the left and right processors of the original one.

Each cell in the proposed array can be considered to have two parts - a computation unit (CU) and a voting unit (VU). The hexagonal array can be modelled as multiple stages of linear arrays and the output of one stage is fed as input to the following stage. The basic assumption of 3DFT is that no two computation sites, on any stage, out of the three involved in doing the same computation can be faulty simultaneously; otherwise the fault cannot be tolerated. Also a fault in the decoder or the OR gate can be modelled as a faulty voting unit.

2.6.1 Faults tolerated in a single stage:

1. If only one CU out of every three consecutive ones is faulty, the fault is masked off by the voters and will result in having three correct outputs that will propagate to the next stage. This type of fault is tolerated.

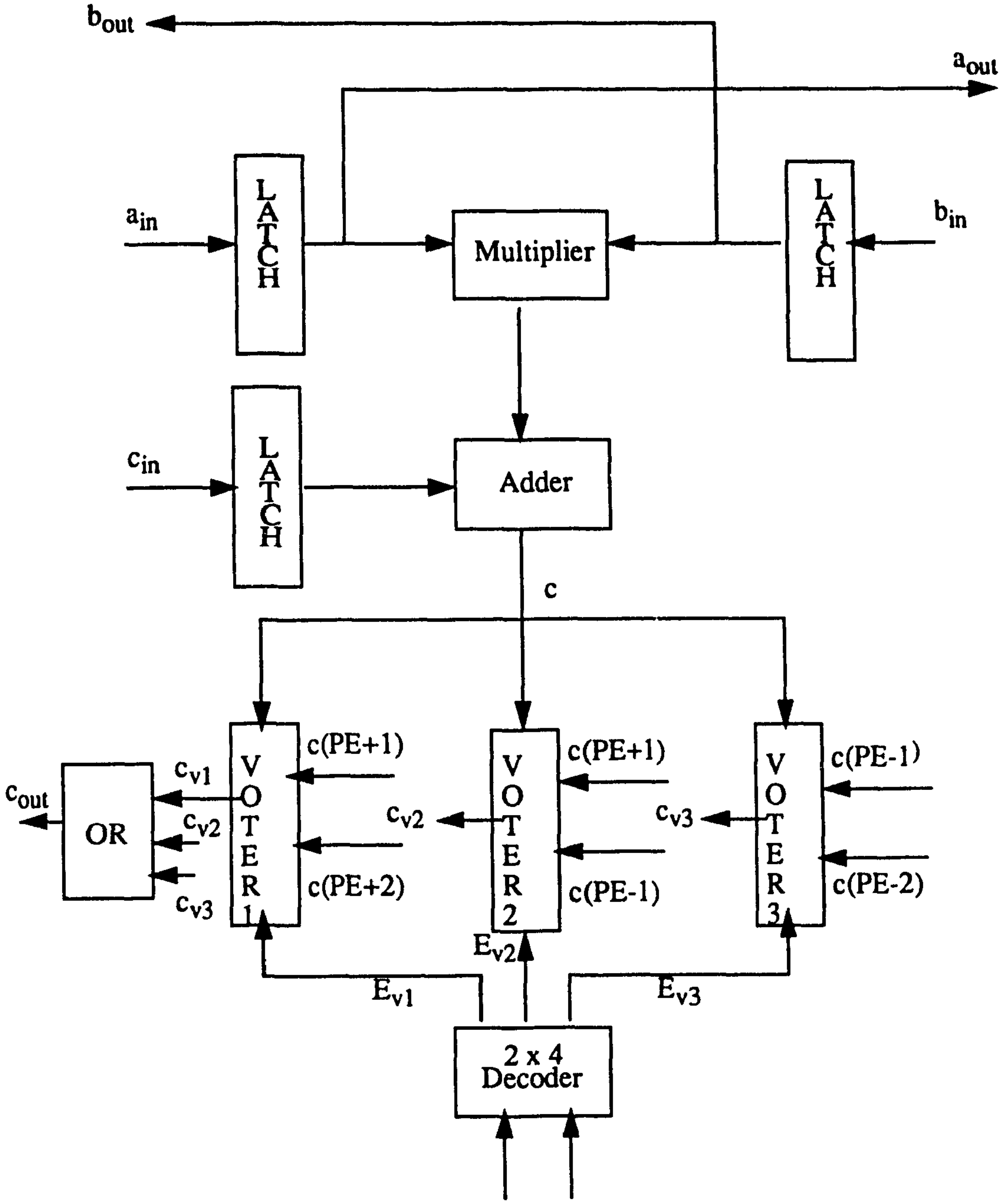


Fig 7. PE design for the matrix multiplication problem.

2. If only one VU out of every three consecutive ones is faulty, the output of this voting unit is faulty. The fault may be tolerated and masked off depending on the fault distribution of the following stage. This will be discussed in more detail in the next section.
3. If only one VU and one CU out of every three consecutive PEs are faulty, this will lead to a faulty output from the PE which have the faulty voting unit only. This will be same as case 2 above.

2.6.2 Faults tolerated in two stages:

Notice a failure in two CU or two VU out of three consecutive PEs on any of the stages cannot be tolerated by the 3DFT approach.

2.6.2.1 Both links and voting units are fault free.

This guarantees that inputs and outputs from each stage are fault free. Only one CU in the first stage and one in the following stage in any three consecutive cells can be faulty.

Assuming that for every stage PEs with numbers $(\alpha + 3\rho)$ are faulty, where $\alpha = 1$, 2 or 3 and $\rho = 0, 1, 2, .., \lceil (W_{max} - 1) / 3 \rceil$, then the maximum number of tolerated faults can reach up to one third of the array size.

Assume that the size of the original hexagonal array is $(W_{max} \times W_{min})$, then the size of the fault-tolerant hexagonal array will be $(W_{max} + 2) \times W_{min}$. The maximum number of tolerated faults per row is (1 + quotient $((W_{max} + 1) / 3)$) and the maximum number of tolerated faults in the array is (1 + quotient $((W_{max} + 1) / 3)) \times W_{min}$.

Example

Assume that CU of both PE 1 and PE 4 are faulty in Fig 2.a. The outputs of PEs 1, 2 and 3 will be the output of the voting unit of these PEs. The faulty result calculated by PE 1 will be masked off. Then PEs 4 and 5 will receive the correct input. Same scenario will be repeated in PEs 4, 5 and 6.

2.6.2.2. Only links are fault free.

Different scenarios may take place:

1. If only one CU in the first stage and one CU in the following stage are faulty, then the scenario in section 2.6.2.1 will take place.
2. Faults in one CU of the first stage and one VU of the second stage. This will result in having one of the outputs (of the faulty VU) of the second stage being faulty. This type of faults can be masked off in the following stage depending on the fault distribution of this stage.
3. Faults in one VU of the first stage and one VU of the second stage. This will result in having one of the outputs (of the faulty VU) of the second stage being faulty. This type of fault can be masked off in the following stage depending on the fault distribution of this stage.
4. Faults in one VU of the first stage and one CU of the second stage.Two cases should be considered:

   (a) If the VU in the first stage is faulty and the corresponding computation in the next stage is also faulty, the faults can be tolerated. The three outputs of this final stage will be correct. For example, if the VU of PE 2 is faulty and the CU of PE 5 is faulty in Fig 2.a, the final output from PEs 4, 5 and 6 will be correct.

   (b) If a VU in the first stage is faulty and any of the other two CUs (other than the corresponding one) of the following stage is faulty, the fault cannot be tolerated. This is equivalent to having two successive faulty PEs on the second stage, which violates the condition to tolerate the faults (only one of every three consecutive PEs can be faulty). For example, assume that the VU of PE 2 is faulty and the CU of PE 4 (Fig 2.a) is faulty. This type of fault is not tolerated. The fault in the VU of PE 2 will feed the CU of PE 5 with wrong input, which in turn will result in a false output of the voting unit because the computation unit of PE 4 will also give a wrong output. When the voting is done the outputs from PE 4, 5 and 6 will be false, and this false output will be propagated to the next stage.

2.6.2.3 Links are faulty.

1. Faults in output links. This would be equivalent to having the VU of the first stage and the corresponding CU of the second stage being faulty. This fault can be tolerated as discussed in section 2.6.2.2.
2. Faults in input links. This would be equivalent to having the CU which have that link as input being faulty.

Example

Assume that the link between PE 1 and PE 2 (in Fig 2.a) is faulty (either the "a" path or "b" path), this would be equivalent to having computation unit of PE2 faulty.

2.7 Advantages of 3DFT

The following advantages are for the Min-Overhead CRC (dt = 0, dx = ± 1, dy = 0). Other configurations will have a major impact on the increased array size. The original hexagonal array size is assumed to be (W_{min} x W_{max}), where W_{min} and W_{max} are the minimum and the maximum bandwidths of the multiplied matrices.

1. The overhead in hardware using the 3DFT approach is equal to (2 x $W_{min)}$), and the rate of increase in hardware compared to the original array size is (2 / W_{max}), which means that the increase in hardware is minimal with the increase of the array size.
2. Both permanent and transient faults can be masked off by the technique if the fault assumption is satisfied.
3. The time complexity is the same as for the original problem since only the idle cycles of the processors are used to do the redundant computations.

2.7.1 Comparison between Reddy's Approach for LU-Decomposition and 3DFT

Kim and Reddy [KimJ87a] remapped the LU-decomposition problem from a hexagonal systolic array to a rectangular bidirectional systolic array. For detailed explanation about this algorit..m see [Mead80]. They also proposed a scheme to detect and locate permanent faulty cells. They adopted the concurrent error detection scheme called Comparison with Concurrent Redundant Computation (CCRC) proposed by Gulati and Reddy [Gula86].

Scope of CCRC: VLSI implementation of systolic algorithms can be classified into two main categories. The first includes those implementations in which data or (sub) results stay in specific cells during the entire course of computation. and the second consists of realizations in which data as well as (sub) results keep moving from cell to cell during computation. The second category of implementations fall under the scope of the proposed CED approach called CCRC.

Proposed Method: CCRC is based on the observation that there is inherent spatial redundancy in the array which could be exploited to perform a concurrent redundant computation. Two computations can be performed in different regions of the array. Then, at the time when the computational wavefront of the required computation reaches a faulty cell, the shadow (redundant) computation reaches a fault-free cell and this computation would be confined to a fault-free region of the array; and thus, a comparison of the corresponding results would lead to the detection of the fault. For duplicating any computation, the two cells involved must receive the same inputs. CCRC proposed earlier can only detect faults but cannot locate faults. By adding additional logic, a single faulty cell among three consecutive cells can be located from the outcome of two consecutive comparisons under permanent fault assumptions.

Assumptions and Hardware overhead: Kim and Reddy assume that the error detection logic is fault free and that at most one PE in every three consecutive PEs in a row is faulty, which is similar to our assumption in the Min-Processor CRC case. The hardware overhead in their technique to achieve CCRC is one extra column of PEs, three multiplexers which are driven by a modulo 2 counter and the error detection logic required in every cell. The error detection logic includes two comparators, a memory element and an OR gate. If fault location is also desired the following extra elements will be needed for every cell: two extra multiplexers, one memory element and one OR gate.

Our proposed 3DFT would handle this problem more efficiently and easily on the original hexagonal array with the following additional advantages:

1. transients faults are also tolerated, and
2. fault correction by masking the faulty PE is achieved.

2.7.2 Comparison between Cosentino's Approach and 3DFT

Cosentino proposed an approach based on temporal redundancy [Cose88] to detect and correct single faults in linear and rectangular systolic arrays in which the cells retain partial results rather than pass them. The cost of the method is halving the maximum throughput rate of the array. As for hardware overhead, the scheme needs monitoring and correction circuits that can be placed on-chip or off-chip and embody the logic for detecting and correcting errors. In addition, an extra accumulator is added to every PE to accommodate the second calculations.

Assuming an original computation of latency equal to three for a rectangular two data flow systolic array, every PE will have three accumulators to accommodate for the three computations and only one computation unit, according to Cosentino's approach. The output will be flushed after the computations are done and voting circuits on the output of every row will be needed.

To do a fair comparison with 3DFT, we will assume that the redundant computations will be mapped with dx = ± 1. This means that the three computations will be done on the same PE. The problems in this approach are the followings:

1. If any of the computation units fails, the three outputs of this PE will be faulty. The fault cannot be tolerated.
2. If any of the input links fails, this will be equivalent to having the computation unit which has that link as input being faulty. The fault is fatal.
3. If any of the output links fails, this will result in having a faulty output. This fault is also fatal.

Even though we assume that the links may not fail as they are simple wires, the failure of a single computation unit will be disastrous. However, a major advantage of this technique is that the overhead in hardware is only two additional accumulators. While 3DFT has three voters in every PE. In addition, every PE has its own computation unit in the 3DFT approach. Another difference between Cosentino's approach and 3DFT is that the voting in 3DFT is done in every PE, while in Cosentino's technique it is done after the computation is complete.

Cosentino's approach is applicable to a class of systolic arrays whose latency is 2 in which the generated variable must stay within the cell. While, the 3DFT approach is applicable to a class of rectangular 3-data flow systolic arrays whose latency is greater than or equal to three and where the generated variable flows out of the processor. The extra latency required by 3DFT, compared with Cosentino's technique, allows us to mask off the faults on every partial result of a computation as long as only one PE out of every three involved in doing this computation is faulty without having to wait till the whole computation is completed. In addition, the 3DFT technique gives the possibility to tolerate more faults, compared with a single fault in the whole array of Cosentino's technique.

Chapter 3

RECONFIGURING UNIDIRECTIONAL 3-DATA FLOW SYSTOLIC ARRAYS

3.1 Survey of reconfiguration techniques

Reconfiguration techniques are used to address the fundamental problem of fault tolerance, both at production and at run time. The former type includes faults in a wafer or a chip, as soon as it comes off the production line. The latter type occurs during the normal operation of the array. Reconfiguration introduces fault-tolerance in the array and produces a functional array even in the presence of multiple faults.

Reconfiguration techniques can be divided into three specific categories [John89]:

1. *Fabrication-time reconfiguration* which is performed immediately after manufacturing.
2. *Compile-time reconfiguration* which is performed before each use of the array, but not during the normal operations of the array, and
3. *Real-time reconfiguration* which is performed while the array is in operation and continues to provide uninterrupted performance.

This chapter is concerned with the first two types of reconfigurations. Issues concerning compile-time and fabrication-time reconfiguration have many similar attributes. For example, both the schemes do not have any restriction on the amount of time required to perform the reconfiguration, although system availability is extremely important to a real-time application. In addition, external fault-detection algorithms are used to locate the faulty cells in both cases. The fault pattern is then used to find an interconnection pattern that could use the fault-free elements to provide a fully functional target array. Thus, the same reconfiguration algorithm could be used for both fabrication time and compile-time reconfiguration.

The key differences between fabrication-time and compile-time reconfiguration are the time at which the reconfiguration is performed and the reversibility of the reconfiguration decisions. Fabrication-time reconfiguration is usually irreversible or permanent, since the interconnection pattern is established using hard switches or fusible links, so the array, once reconfigured, cannot be changed or modified again. In compile-time reconfiguration, the decisions are reversible, allowing the system to be reconfigured numerous times.

3.2 RC- and RCS-cut approaches for unidirectional 2-Data flow systolic arrays

It is generally assumed that only the computational section of the cells can fail and all other components (registers, interconnections, etc.) in the array are failure-free. For reconfiguration, the computational section of a faulty cell is bypassed and the cell (a pass cell) is made to pass the data without processing. Note that the reconfiguration must bypass all faulty cells and it may also bypass some nonfaulty cells. The reconfiguration is performed by selecting a sequence of cuts. A *cut* is a set of cells such that bypassing them leads to an array with one less data flow path. A cut is *horizontal* (*vertical*) if it leads to the removal of one horizontal (vertical) data flow path. A cut covers the cells contained in it. Thus, for successful reconfiguration, a sequence of cuts covering all the faulty cells should be selected.

In the classical approach [Kore81], for every faulty cell all the cells in the same row or column are taken to be in a cut. Thus, an entire row or an entire column containing at least one faulty cell constitutes a cut, and these cuts are referred to as the RC cuts. In this case, a horizontal cut is also referred to as a row cut and a vertical cut as a column cut. The structure of a cell suitable for this RC-cut approach is shown in Fig.8.a. The classical approach is simple (and leads to easy reconfiguration procedures) but provides poor utilization of nonfaulty cells. In Kung and Lam's approach [KuHT84], the cells in a cut (referred to as RCS cut, S denoting slanted cut) may not be from the same row or column but satisfy the following conditions.

Necessary Condition: A cut must contain one cell per row (vertical cut) or one cell per column (horizontal cut) and the slope of the line connecting successive cells in the cut must be nonnegative.

Sufficient Condition (Reachability Condition): The inclination of the line connecting the cells in the cut between successive columns must be 0 or 45 degrees for horizontal cuts, or 90 or 45 degrees between successive rows for vertical cuts.

The bypassing is done with data rerouting so that the horizontal input may be sent to the horizontal or the vertical output and the vertical input undergoes similar rerouting. Kung and Lam's approach requires more complicated processor architectures and more involved reconfiguration algorithms but it provides a better utilization of nonfaulty cells. A cell architecture supporting the RCS-cut approach is shown in Fig. 8.b.

Lam et al presented a critical study of Kung's approach and the basic row-column elimination approach [LamC89]. Notice that these two approaches have been applied to two directional data flow (one horizontal and one vertical). This thesis extends the RCS-cut approach to cover the case of having a diagonal input and a diagonal output. A cell architecture supporting this case is shown in Fig 8.c.

3.3 RCS-cut approach for unidirectional 3-data flow systolic arrays

The proposed scheme follows Kung and Lam's approach in that the faulty cells are bypassed systolically. The same RCS cuts suggested above for the unidirectional 2-data flow systolic arrays are applied for the unidirectional 3-data flow systolic arrays; that is, the same necessary and sufficient conditions have to be satisfied. Not more than one cut will be selected at a time. Once a cut is identified, data rerouting has to be achieved. It will be assumed, as previously, that only the computational section of the cells can fail and all other components in the array are failure-free. For reconfiguration, the computational section of a faulty cell is bypassed and the cell is made to pass the data without processing.

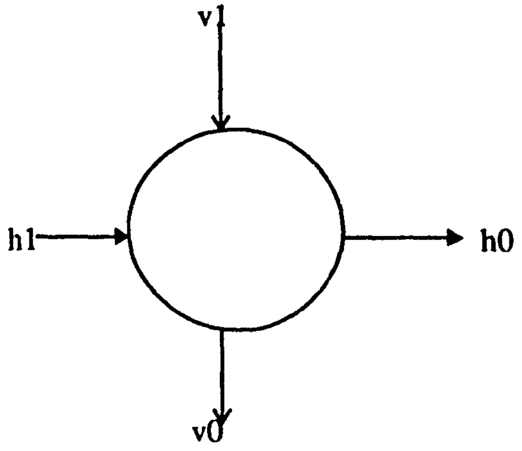


Fig. 8.a Architecture of processing elements for RC-cut approach

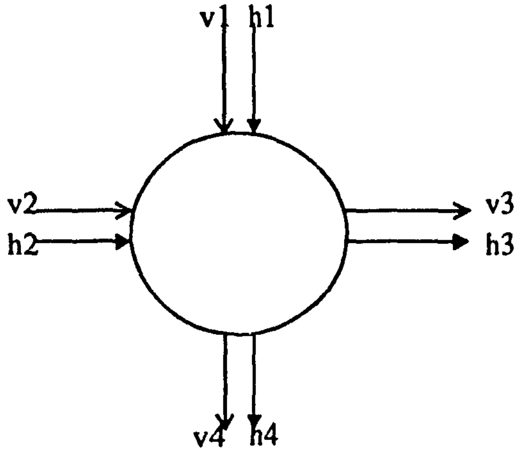


Fig. 8.b Architecture of processing elements for RCS-cut approach

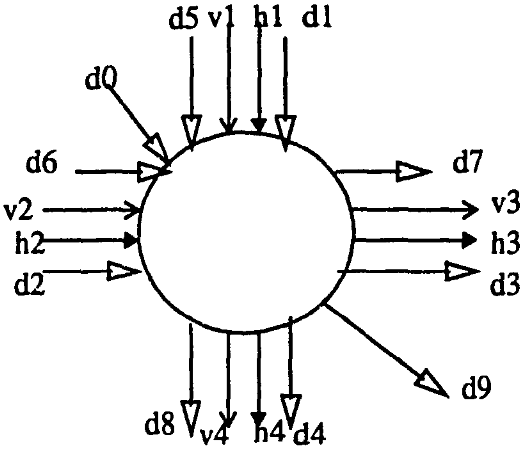


Fig. 8.c Architecture of processing elements for RCS-cut approach with diagonal input and diagonal output.

For either a vertical or a horizontal cut, the flow path for the horizontal and the vertical data will be exactly the same as in the case of 2-data flow. Only, the diagonal path needs to be modified to flow along the horizontal or vertical paths.

3.3.1 Vertical Cut

Generally, the diagonal path will flow diagonally once the computation is performed. Otherwise, it will be redirected towards the right cell along with the horizontal path.

1. Cell in a cut (faulty or good) has horizontal input h_i coming from left and diagonal input d_k coming diagonally. The two inputs will be redirected together to the right cell. Fig.9.a shows the flow path if the right cell is good and Fig.9.b shows the flow path if the right cell is faulty. Notice that in all the following figures, a shaded cell represents a cell in a cut (faulty or good) and a black one represents a faulty cell.
2. Cell in a cut (faulty or good) has the three inputs available in the regular way (horizontal input h_i coming from left, vertical input v_j coming from top and diagonal input d_k coming diagonally). The three inputs will be redirected together to the right cell. If the latter is good, the computation will be performed among the three redirected inputs. As shown in Fig.10.a, the horizontal output flows to the right, the vertical output flows downward and the diagonal output flows diagonally. The other diagonal input to this cell will flow to the right with the horizontal output. If the right cell is faulty, all its inputs will be redirected to the right cell as shown in Fig. 10.b. Notice that when a cell receives two diagonal inputs, the first redirected diagonal has higher priority over the other one when it comes to computation.
3. Cell in a cut (faulty or good) has both horizontal input h_i and diagonal input d_k coming from left. The vertical input is absent. The two inputs will be redirected together to the right cell. Fig.11.a shows the flow path if the right cell is good and Fig.11.b shows the flow path if the right cell is faulty.

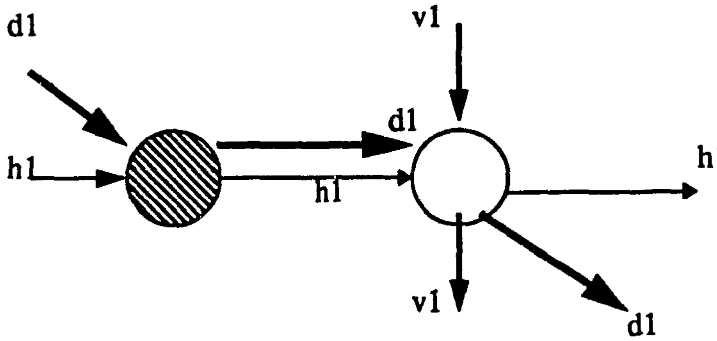


Fig 9.a Vertical cut with outputs reaching a good cell (1)

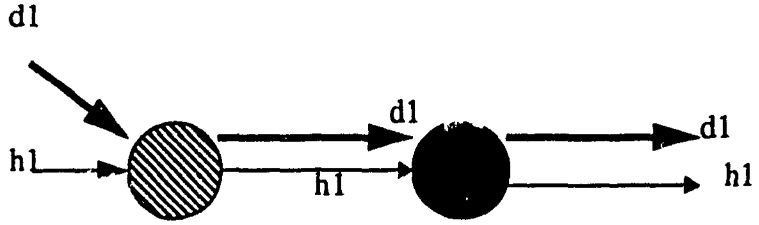


Fig. 9.b Vertical cut with outputs reaching a faulty cell (1)

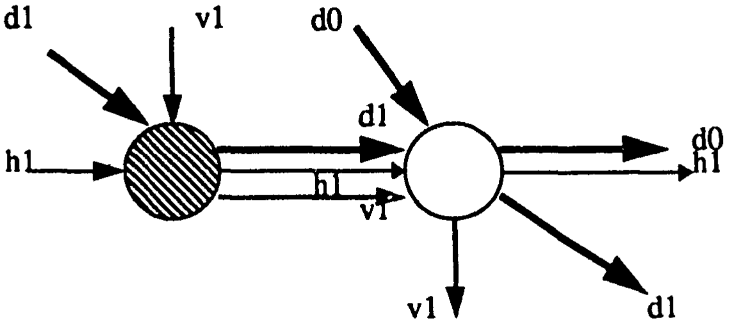


Fig.10.a Vertical cut with outputs reaching a good cell (2)

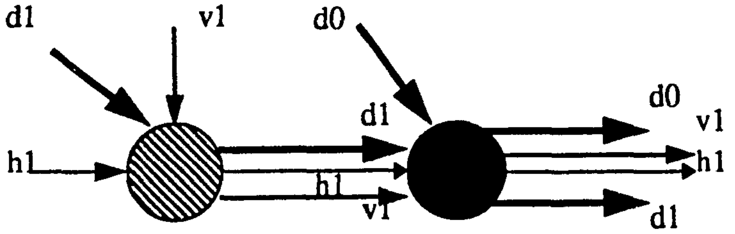


Fig. 10.b. Vertical cut with outputs reaching a faulty cell (2)

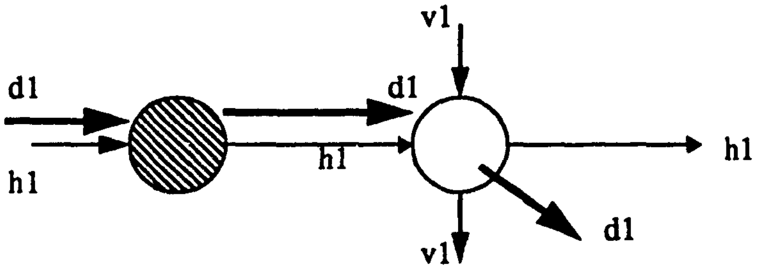


Fig.11.a Vertical cut with outputs reaching a good cell (3)

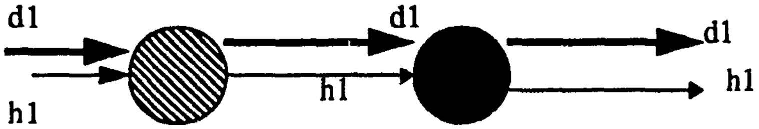


Fig.11.b Vertical cut with outputs reaching a faulty cell (3)

4. Cell in a cut (faulty or good) has two diagonal inputs, one coming from left (redirected) and the other coming diagonally. In addition, both the horizontal and vertical inputs are coming from left. Fig.12.a shows the flow path if the right cell is good and Fig.12.b shows the flow path if the right cell is faulty.
5. Cell in a cut (faulty or good) has two diagonal inputs, both coming from left. In addition, both the horizontal and vertical inputs are coming from left. Fig.13.a shows the flow path if the right cell is good and Fig.13.b shows the flow path if the right cell is faulty.

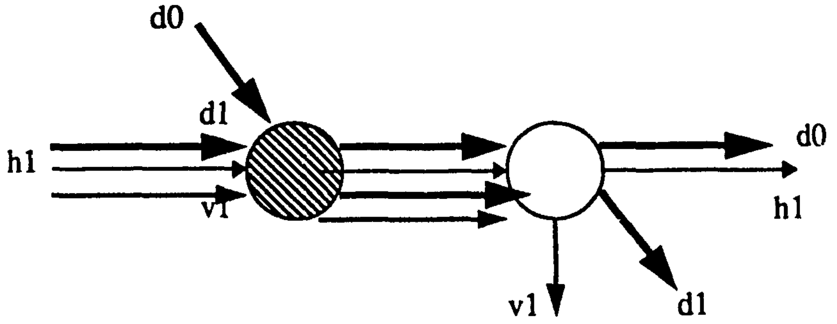


Fig.12.a Vertical cut with outputs reaching a good cell (4)

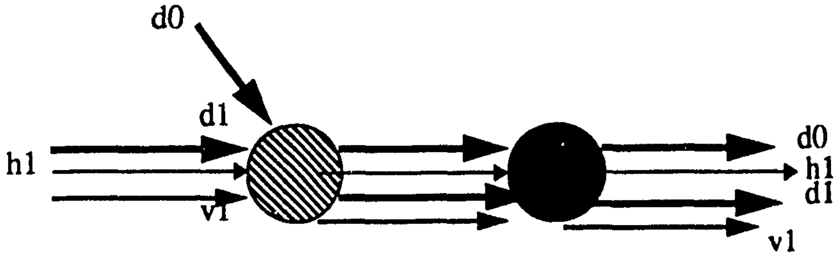


Fig.12.b Vertical cut with outputs reaching a faulty cell (4)

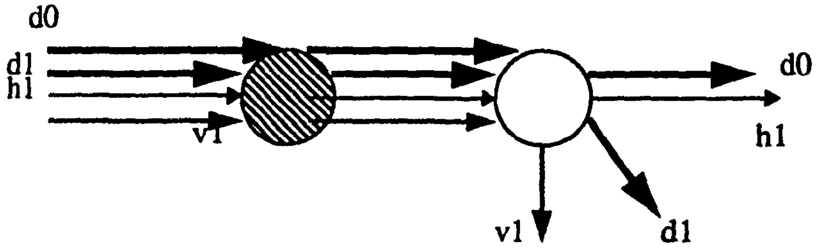


Fig.13.a Vertical cut with outputs reaching a good cell (5)

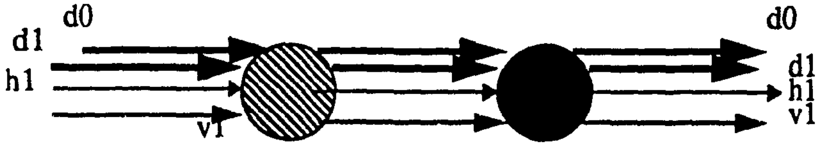


Fig.13.b Vertical cut with outputs reac^h^ng a faulty cell (5)

3.3.2 Horizontal Cut

Generally, the diagonal path will flow diagonally once the computation is performed. Otherwise, it will be redirected towards the bottom cell along with the vertical path.

1. Cell in a cut (faulty or good) has vertical input v_j coming from top and diagonal input d_k coming diagonally. The two inputs will be redirected together to the bottom cell. Fig.14.a shows the flow path if the bottom cell is good and Fig.14.b shows the flow path if the bottom cell is faulty.
2. Cell in a cut (faulty or good) has the three inputs available in the regular way (horizontal input h_i coming from left, vertical input v_j coming from top and diagonal input d_k coming diagonally). The three inputs will be redirected together to the bottom cell. If the latter cell is good, the computation will be performed among the three redirected inputs. As shown in Fig.15.a, the horizontal output flows to the right, the vertical output flows downward and the diagonal output flows diagonally. The other diagonal input to this cell will flow downward with the vertical output. If the bottom cell is faulty, all its inputs will be redirected to the bottom cell as shown in Fig. 15.b.
3. Cell in a cut (faulty or good) has both vertical input v_j and diagonal input d_k coming from top. The horizontal input is absent. The two inputs will be redirected together to the bottom cell. Fig.16.a shows the flow path if the bottom cell is good and Fig.16.b shows the flow path if the bottom cell is faulty.
4. Cell in a cut (faulty or good) has two diagonal inputs, one coming from top (redirected) and the other coming diagonally. In addition, both the horizontal and vertical inputs are coming from top. Fig.17.a shows the flow path if the bottom cell is good and Fig.17.b shows the flow path if the bottom cell is faulty.

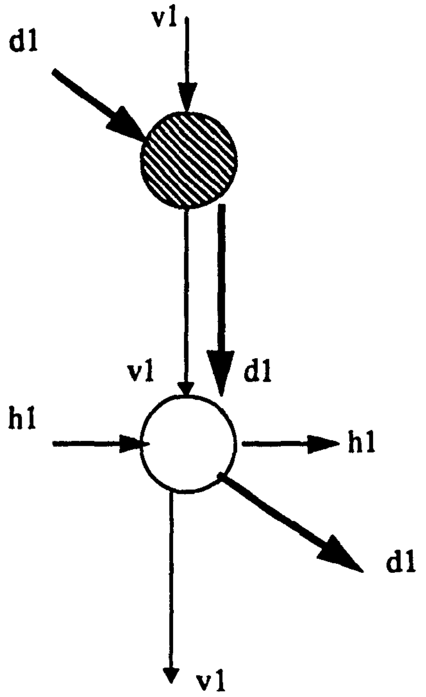


Fig 14.a Horizontal cut with outputs reaching a good cell (1)

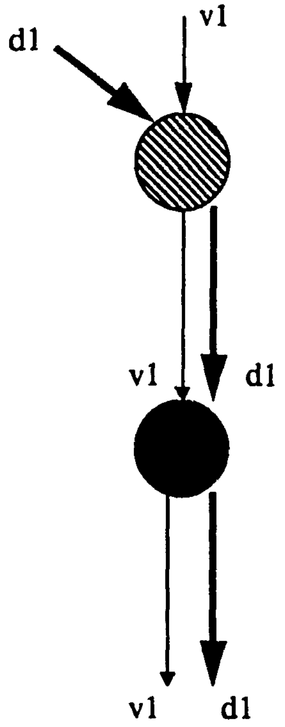


Fig.14.b Horizontal cut with outputs reaching a faulty cell (1)

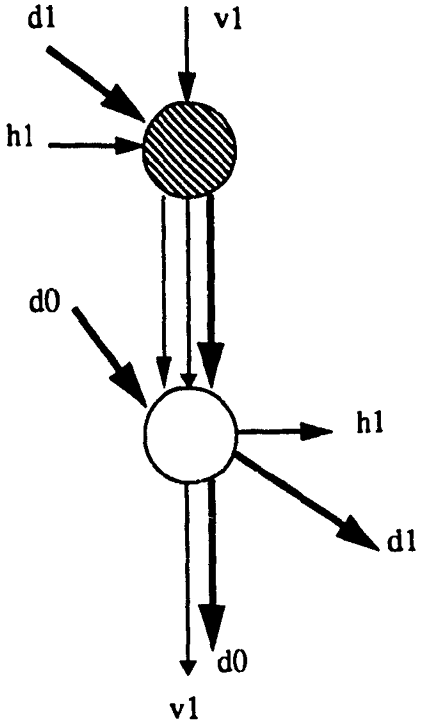


Fig.15.a Horizontal cut with outputs reaching a good cell (2)

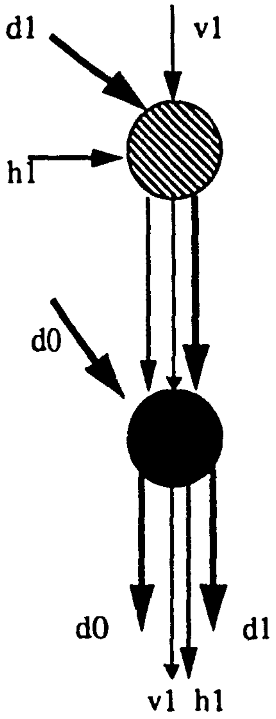


Fig. 15.b. Horizontal cut with outputs reaching a faulty cell (2)

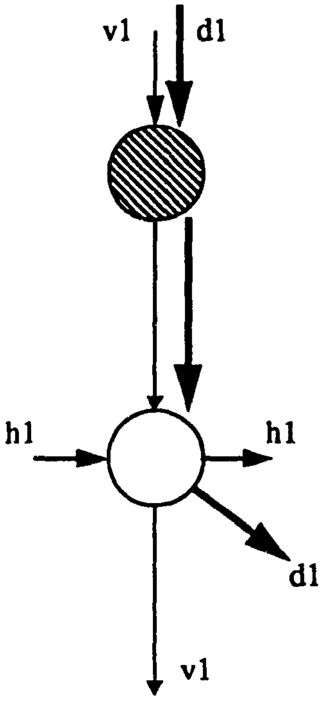


Fig 16.a Horizontall cut with outputs reaching a good cell (3)

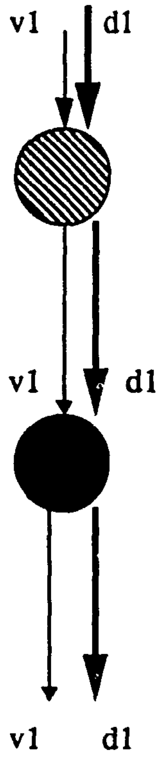


Fig 16.b Horizontal cut with outputs reaching a faulty cell (3)

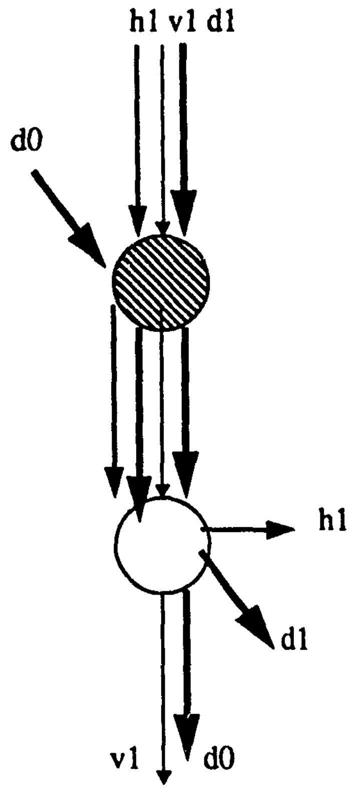


Fig. 17.a Horizontal cut with outputs reaching a good cell (4)

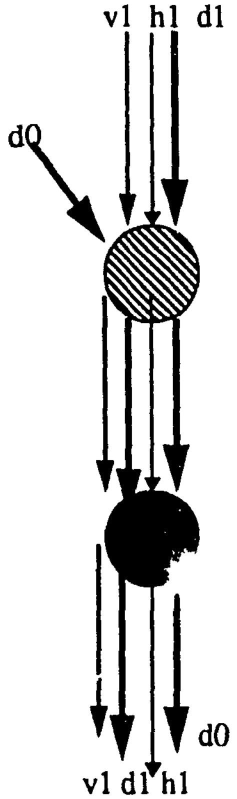


Fig.17.b Horizontal cut with outputs reaching a faulty cell (4)

5. Cell in a cut (faulty or good) has two diagonal inputs, both coming from top. In addition, both the horizontal and vertical inputs are coming from top. Fig.18.a shows the flow path if the bottom cell is good and Fig.18.b shows the flow path if the bottom cell is faulty.

EXAMPLES

1. Fig. 19 shows a reconfigured hexagonal array with two vertical cuts. It illustrates the redirection of two diagonals reaching a good cell
2. Fig. 20 shows another example of a reconfigured hexagonal array with vertical cuts. It illustrates the redirection of two diagonals reaching a faulty cell.
3. Fig. 21.a shows a 5 x 5 hexagonal array with both horizontal and vertical cuts. Fig. 21.b shows the array after applying the vertical cut and Fig.21.c shows the array after applying the horizontal cut.

4 An example of a 2 x 2 matrix multiplication on a hexagonal array with a vertical cut is presented in Fig.22.a. Snapshots of the different computations are shown in Fig. 22.a ı 22.f.

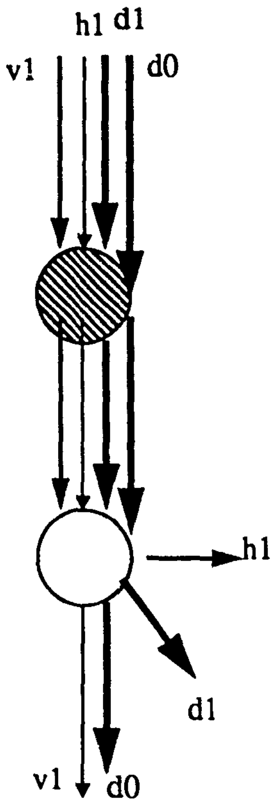


Fig.18.a Horizontal cut with outputs reaching a good cell (5)

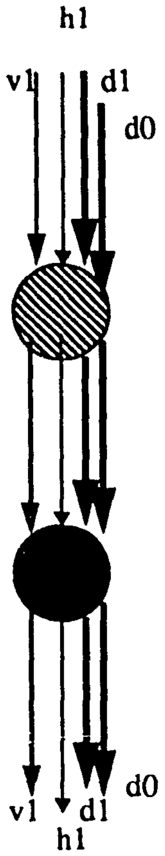


Fig.18.b Horizontal cut with outputs reaching a faulty cell (5)

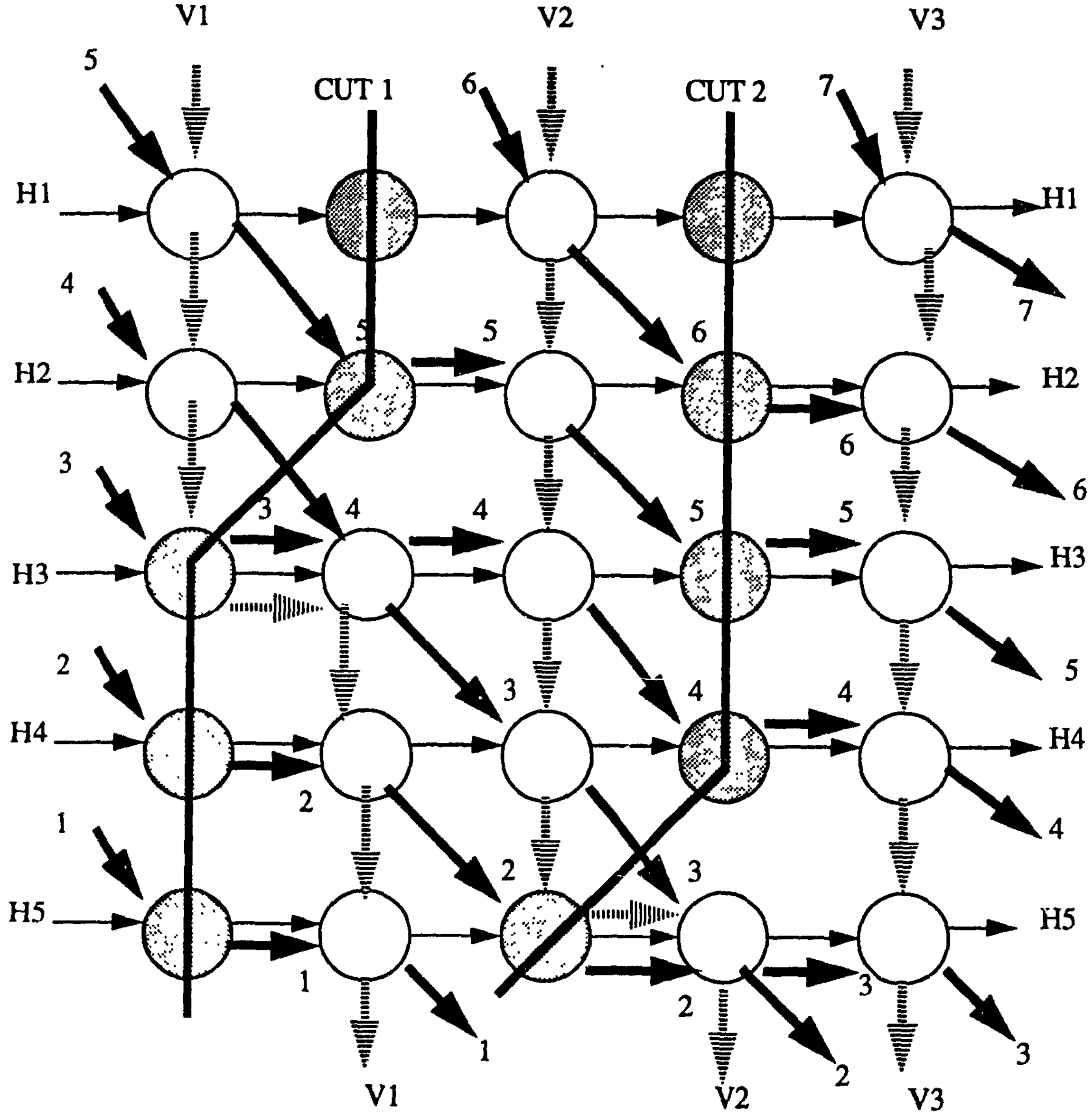


Fig. 19 A reconfigured array with 2 vertical cuts(1)

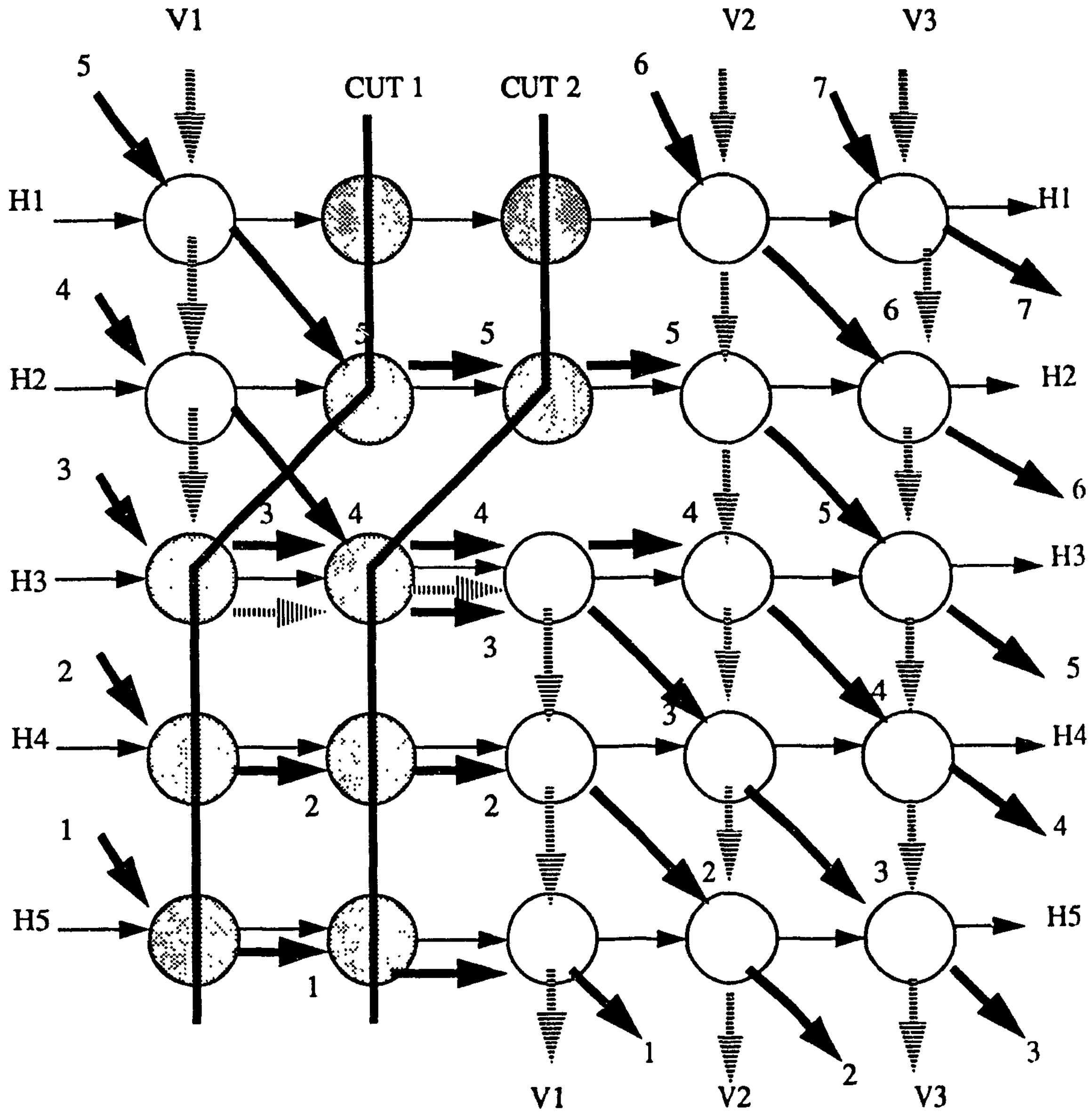


Fig. 20 A reconfigured array with 2 vertical cuts (2)

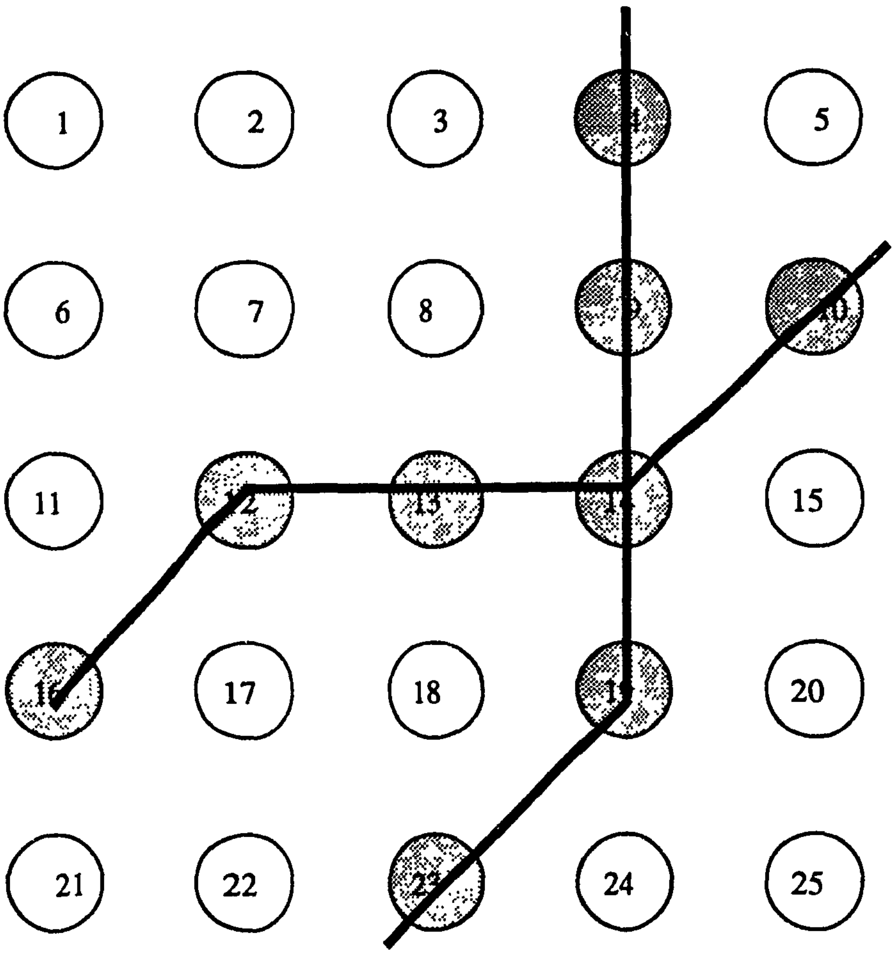


Fig. 21.a A 5 x5 hexagonal array with a vertical cut and a horizontal cut and failures circles are hashed.

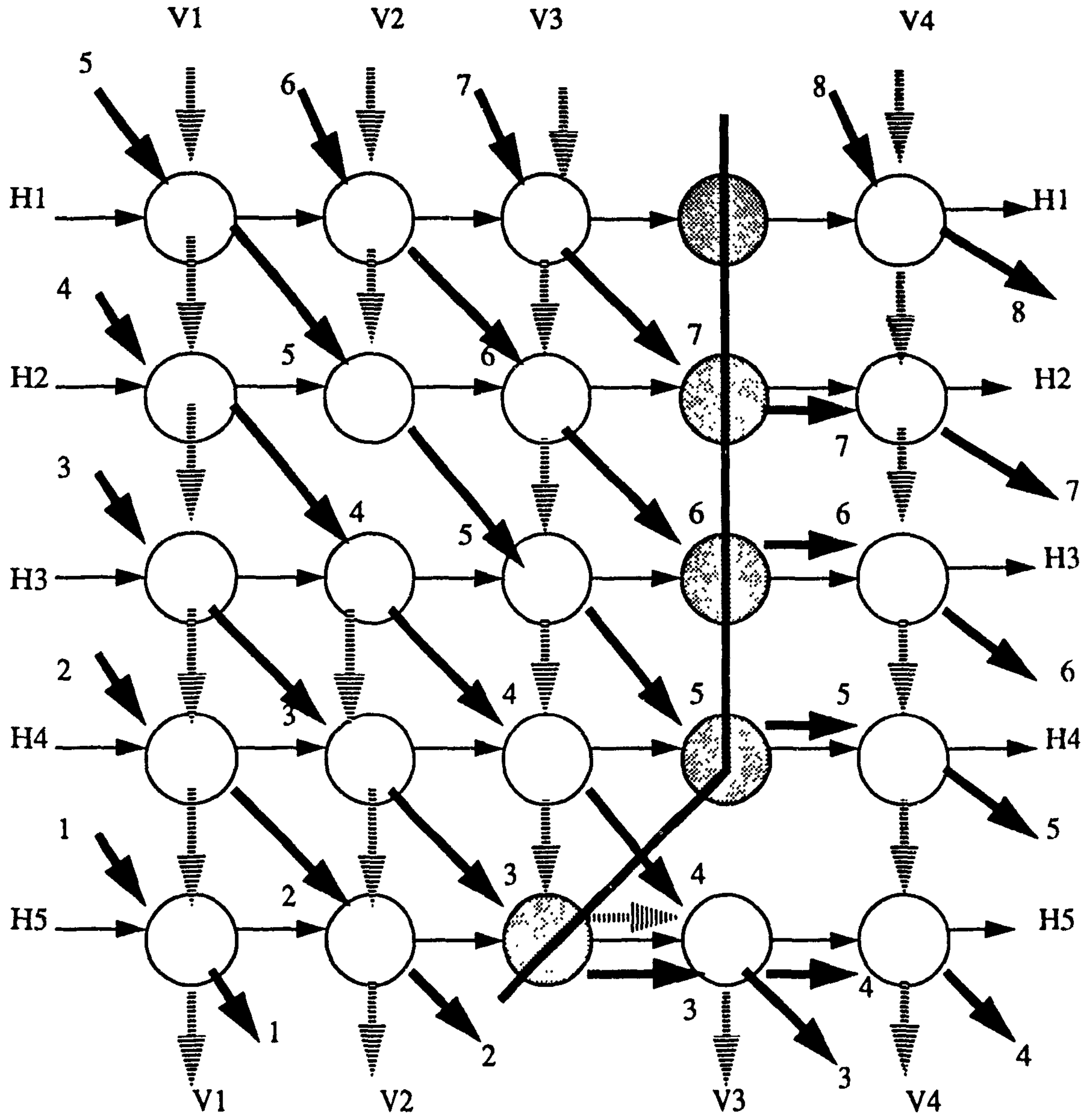


Fig. 21.b The reconfigured array after applying the vertical cut.

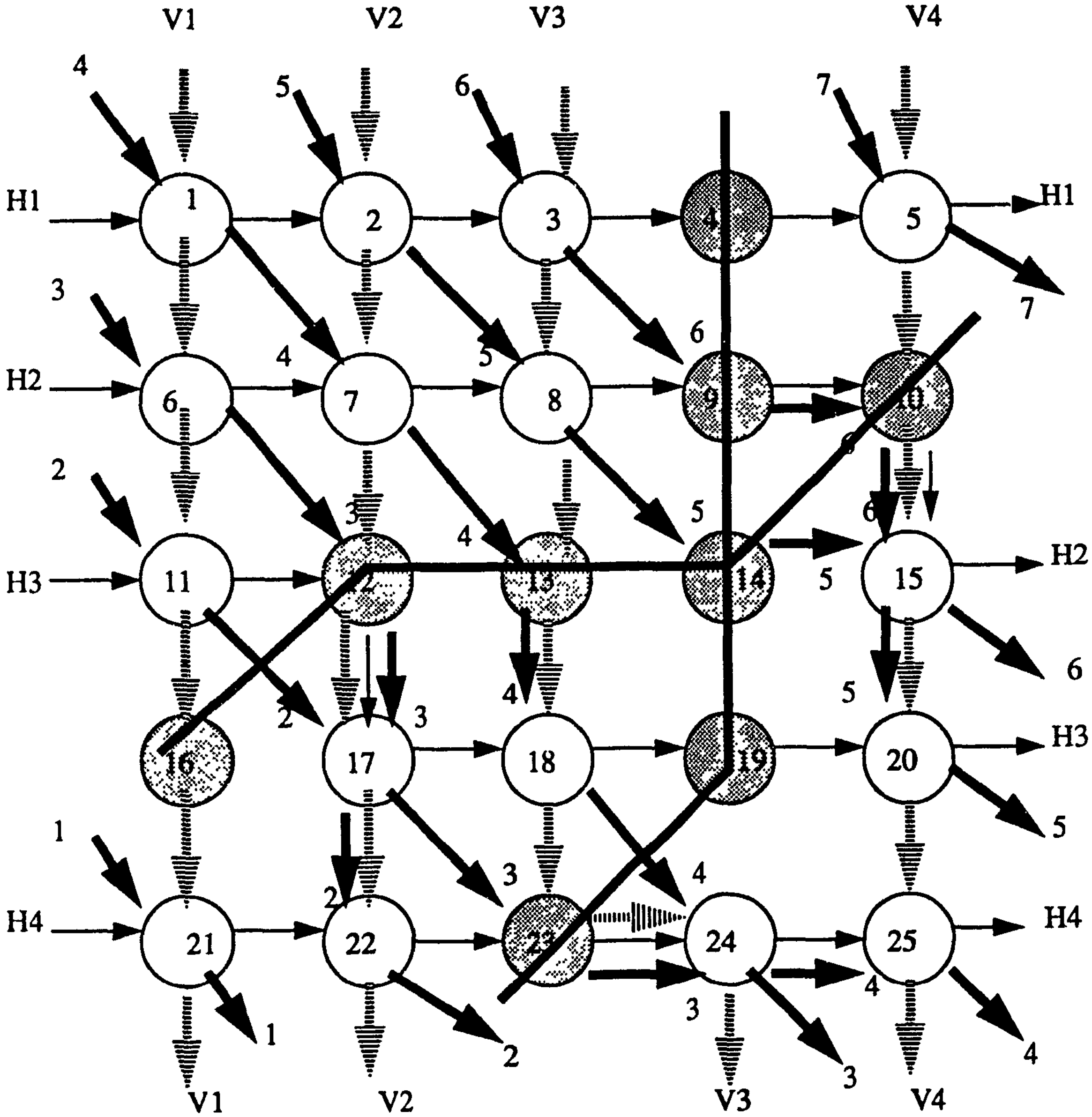


Fig. 21.c The reconfigured array after applying the horizontal cut.

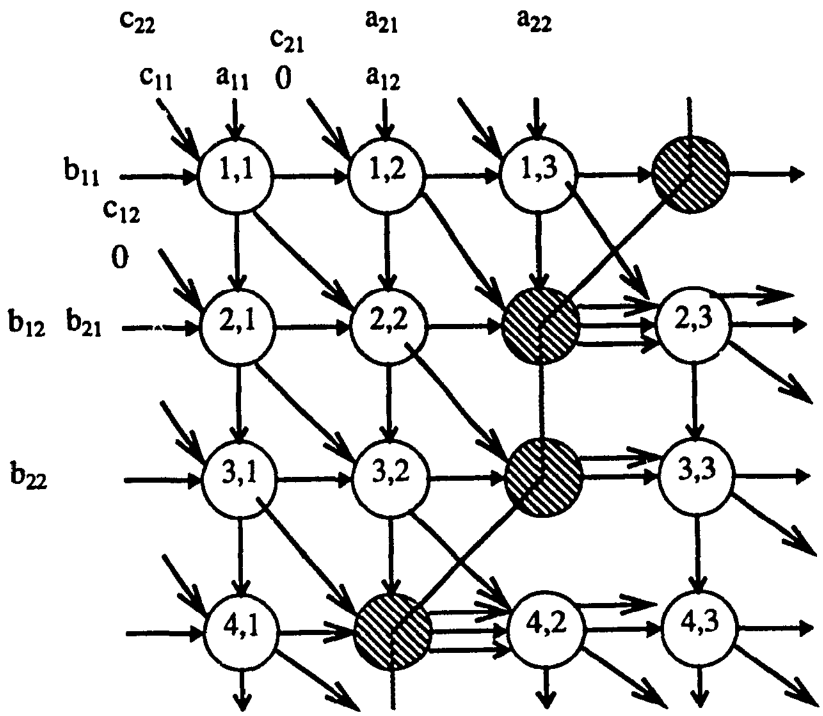


Fig 22.a A 2 x 2 matrix multiplication using a hexagonal array.

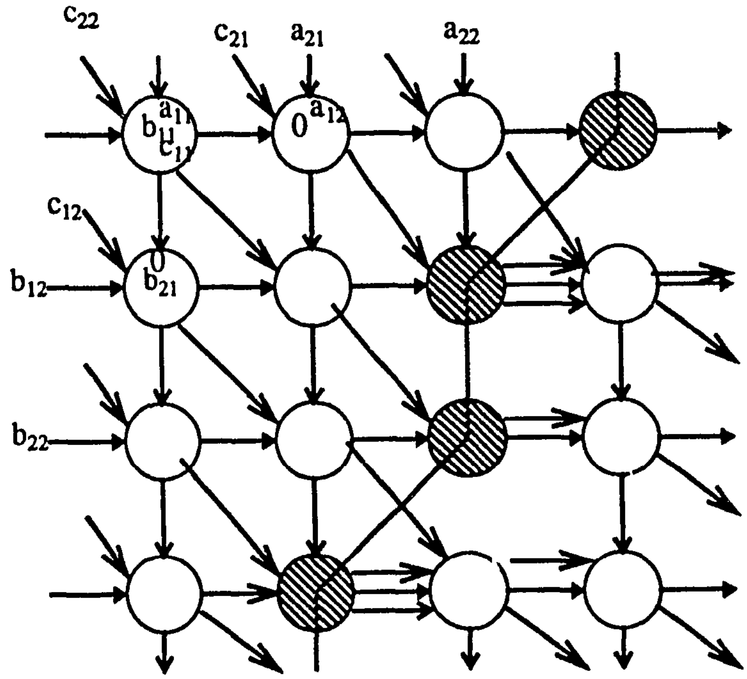


Fig 22.b

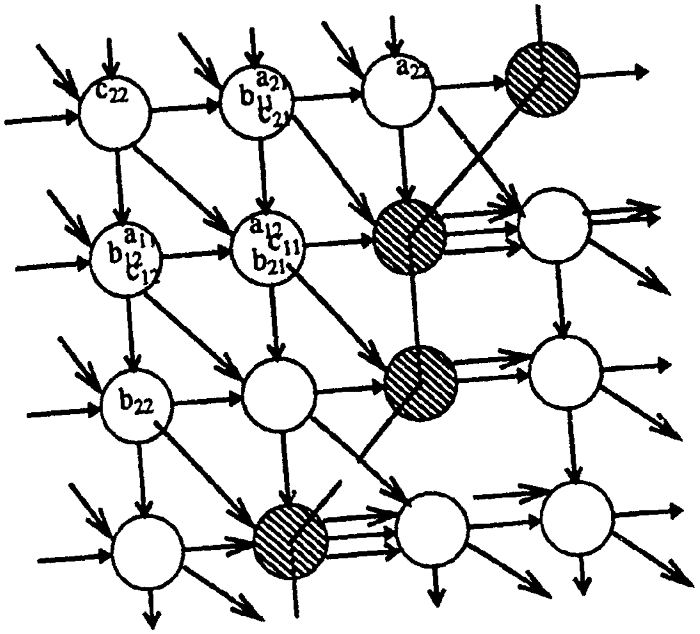


Fig 22.c

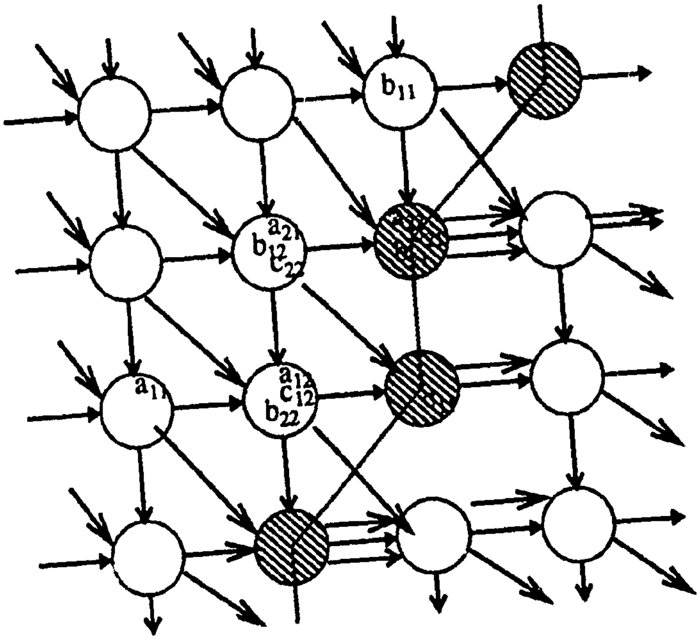


Fig 22.d

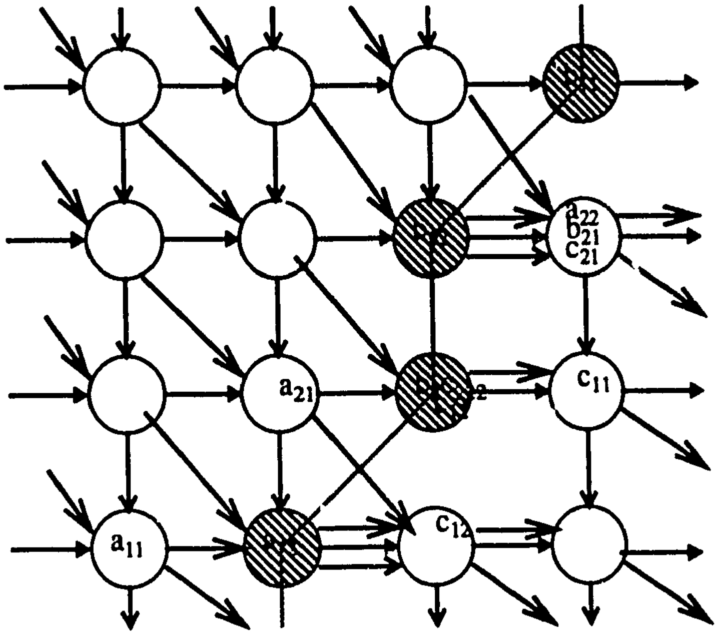


Fig 22.e

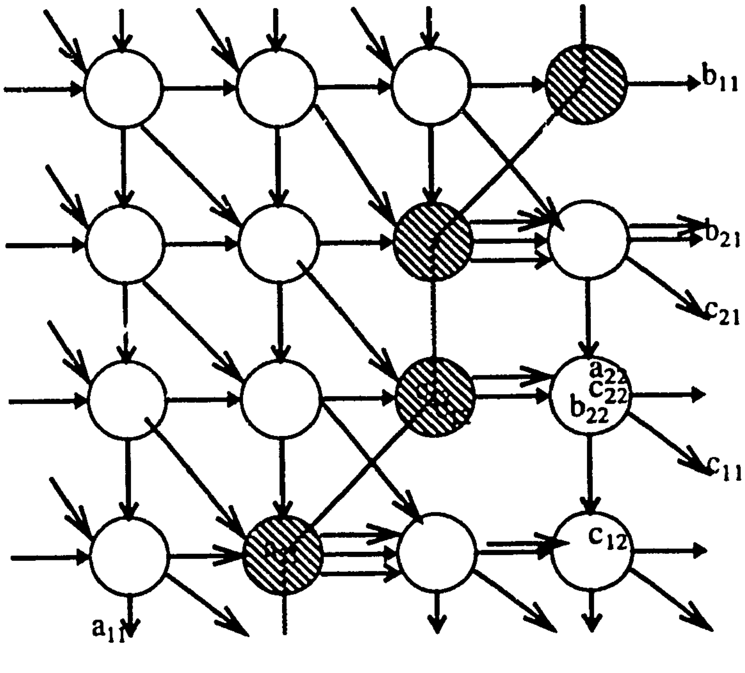


Fig 22.f

3.4 Comparison of restructuring techniques for Hexagonal Arrays

3.4.1 Gordon's Technique (GT)

Gordon et al [Gord87] suggested a scheme for fault-tolerance in hexagonal arrays. When some PEs or connections become faulty, the other PEs were restructured into a hexagonal array (of smaller size). Fault tolerance is achieved in two basic stages, the *testing stage* and *the reconfiguration stage*. In the testing stage, the PEs test their neighbors and themselves, in order to identify faulty PEs or connections. In the reconfiguration stage, the PEs with neighboring faults turn into *connecting elements* (CEs) and initiate messages which turn some other PEs into CEs. These CEs cease to perform processing per se and behave like connectors between pairs of neighboring PEs. Each remaining PE is not aware of the presence of the CEs and continues to communicate with six neighbors as before the reconfiguration occurred, using the same links as it did before (i.e., the neighbor in a given direction is still accessed in that direction). It is possible, however that some of its neighbors are not physically the same as before, and the PE reaches them through some CEs.

The main advantage of this approach is that it makes the restructured array (following the identification of the faulty PE or communication link) transparent to the various algorithms utilizing the hexagonal array. Assuming an initial (n x n) Hexagonally connected array(HCA), if p connections and q PEs become faulty in sequence, then the resulting configuration will contain an (n -p - 2q) x (n - p - 2q) HCA. In addition, the approach does not have any underlying failure-free assumption as many other schemes (like switching elements and communications links are almost failure free and only processors can fail). One major disadvantage is that the utilization would be poor.

Assuming only q PEs may fail for an original (n x n) HCA, that is, the resulting array size will be (n - 2q) x (n - 2q). The number of unused good PE equals 4q (n - q) which can be approximated to 4n, implying poor utilization of good cells.

3.4.2. Kumar's Approach

Recently, Kumar and Agrawal [Kuma91] introduced a design for compile-time reconfigurable hexagonal systolic arrays. A new concept of processing element reachability has been introduced to provide the basis of their reconfiguration algorithm. The reachability R of a processor is defined as the maximum number of PEs to which a PE can directly route its data to. Higher the reachability, higher would be the survivability of the array. R is defined for the three emanating data paths in a hexagonal systolic array: horizontal, vertical and diagonal.

The fault assumption is that only PEs can fail but not the links. Presence of spare rows and spare columns is also assumed. In case of multiple faults, Kumar and Agrawal's reconfiguration approach keeps a fault count for each row of the (N + SR) x (N + SC) original array (where SR and SC are the number of spare rows and spare columns, respectively) in nondecreasing order and always bypasses the first SR rows with maximum number of faults. Thus, the resulting N x (N+SC) array has minimum possible number of faults. This causes much higher utilization of available good cells. Once an N x (N+SC) array is formed, it is then used to form an N x N fault free logical array.

The major disadvantage of this approach is that the reconfiguration depends heavily on the value of reachability R. To overcome the problem, the reachability R should be increased but this will also means longer communication lines which is not desirable. The approach would be suitable if the faulty cells are sparsely distributed in the array rather than clustered. The performance of the reconfiguration approach depends heavily on the fault distribution.

Fig. 23 shows an example of a reconfigured hexagonal array with a horizontal cut. Kumar's approach will fail to reconfigure such an array due to the presence of five consecutive faulty PEs (if the reachability is limited to 5).

3.4.3 RCS-cut approach for hexagonal arrays

In contrast to the previous approach which will behave better in case of sparsely

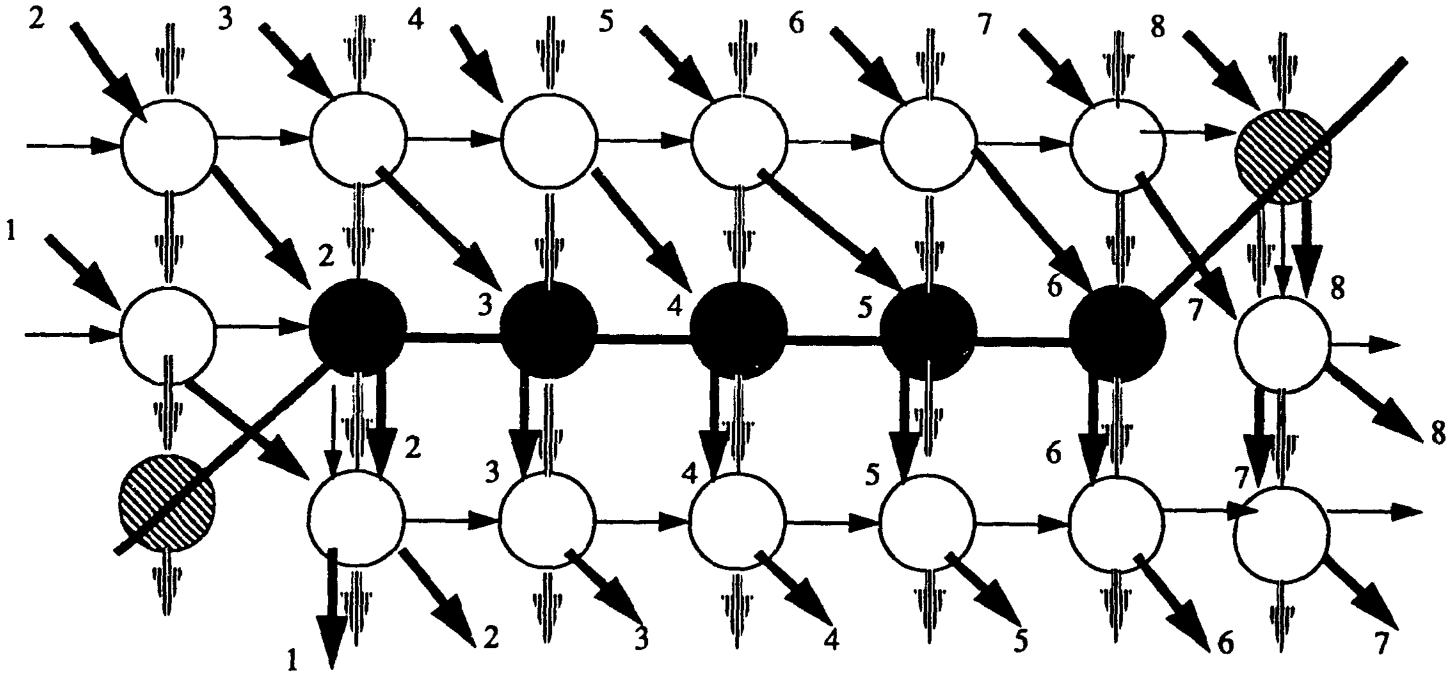


Fig.23 A reconfigured 7x2 hexagonal array with a horizontal cut

distributed faulty cells, the RCS-cut approach would handle better the case of clustered faults (successive faulty cells in a row, column or diagonally). The disadvantage of the proposed RCS-cut approach is that the PE needs nine inputs and nine outputs.

On the other hand, it provides the possibility to reconfigure the array without any restrictions on horizontal, vertical or diagonal reachability, like Kumar's approach. Also, it gives higher utilization of good PEs compared with Gordon's approach.

Assuming only one PE failed for an original (n x n) HCA, the resulting array size will be (n - 1) x n if we apply the RCS cut approach. The number of unused good PE equals (n - 1), which can be approximated to n, compared with 4n unused good cells for Gordon's technique.

Chapter 4

DYNAMIC RECONFIGURATION FOR UNIDIRECTIONAL 3-DATA FLOW SYSTOLIC ARRAYS

4.1 Introduction

As discussed previously, one approach to achieve fault-tolerance is by masking off the faults using the conventional TMR method [KimJ85] as in our approach (3DFT) presented earlier. One drawback of this method is that only limited classes of fault patterns can be tolerated. Another approach to handle this problem is the run-time dynamic reconfiguration which configures the processing array based on local information only. Reconfiguration can be done automatically without interfering with the computation.

This chapter presents a distributed reconfiguration scheme that can tolerate permanent as well as transient faults. The proposed scheme follows Kung and Lam's [KuHT84] approach in that the faulty cells are bypassed systolically. The reconfiguration algorithm uses the data-driven concept, that is, instead of being assigned a fixed path, the data tokens will find their own ways through the processing array. When a fault is detected, the concerned data tokens are re-routed to an adjacent cell to retry the computation. Consequently, some rippling effect may be imposed on the successor cells. So long as there are enough spare cells to replace the faulty ones, a correct computation can be obtained. To achieve this, the systolic array should possess the following two capabilities.

1. *Concurrent fault detection:* The processing cell should be able to determine concurrently with the normal operation whether the performed computation is correct or not.
2. *Self-reconfiguration:* The systolic array should manage its own reconfiguration

without global knowledge about the fault distribution. Reconfiguration is done locally at the cells and can be changed at any time, subject to local status detected.

Pao [PaoD87, LiHF87] presented a distributed reconfiguration approach based on local invariants technique. The re-routing of data flow paths is based on local information only, and can be done without interfering with the computation. He presented designs of two self-reconfigurable uni-directional 2-data flow systolic arrays. In the first design, both the horizontal and vertical paths are reconfigurable, whereas in the second design, only the vertical paths can be reconfigured.

He has also extended the concept to universal systolic arrays in which the diagonal paths are fixed to flow diagonally but the horizontal (vertical) paths are reconfigurable. In this chapter, a dynamic reconfiguration approach for universal systolic arrays in which all the three paths are reconfigurable is presented.

4.2 Distributed Reconfiguration Approach

Distributed reconfiguration is characterized by local d`cision making at individual processing cells. No global knowledge of the fault distribution is required in restructuring the array. A reconfiguration is successful if an embedding of the required array is constructed in the faulty array; otherwise it is unsuccessful.

In a universal systolic array, three data streams, namely, horizontal (H_1, H_2,..., H_m), vertical (V_1, V_2,..., V_n) and diagonal (D_1, D_2,..., D_{m+n-1}), meet at a cell. The horizontal paths are numbered from top to bottom and the vertical paths are numbered from left to right and the diagonal paths are numbered bottommost diagonal to topmost diagonal.

In this distributed reconfiguration approach, the routing of data is based only on local information and can be changed at any time subject to local status detected. No fixed path is assigned to data tokens. Alignment of the data tokens to arrive at the active processing cells is ensured by an invariant technique. A set of local invariants of the reconfiguration algorithm is identified. Intelligence is then incorporated into the processing cells to enforce these invariants.

A path, P1, is said to have crossed another path, P2, if it is extended from one side of P2 to the other side and vice versa. The dynamic routing in a self-reconfigurable array is obtained by enforcing the following two invariants:

1. Two successive horizontal (vertical) (diagonal) data paths H_i and H_{i+1} (V_j and V_{j+1}) (D_k and D_{k+1}) do not cross. Touching at a cell is, however, allowed.
2. The data in horizontal path H_i, in vertical path V_j, and in diagonal path D_k do not cross until they have been processed.

The above two invariants and the following theorem are direct extensions of the original work in [PaoD87].

Theorem 4.1

If an M x N array is reconfigured (with data re-routing) into a smaller array of size m x n in such a way that the two invariants are satisfied, and every horizontal path H_i (i =1,2,...,m) has crossed every vertical path V_j and diagonal path D_{m+j-i} (j = 1,2,...,n and (m +j -i) is greater or equal to 1), and vice versa, then the reconfiguration is successful.

Proof

Since the horizontal path H_i does not cross H_{i+1} (invariant 1), the vertical path V_j cannot cross H_{i+1} before crossing H_i and the diagonal path D_{m+j-i} also cannot cross H_{i+1} before crossing H_i. Similarly, the diagonal path D_{m+j-i} cannot cross V_{j+1} before crossing V_j. When V_j crosses H_i and D_{m+j-i} at a common good cell, the required computation among them should have been completed (invariant 2).

If the reconfiguration is successful, data on V_j will meet and be processed with data on the horizontal and diagonal paths (H_1, D_{m+j-1}), (H_2, D_{m+j-2}),(.....,....), (H_m, D_j) successively in some common good cells.

The same argument applies to the horizontal and diagonal paths. Q.E.D

4.3 Cell Architecture

There is one processing unit in each cell together with a self-testing circuit. The self tester will check the result of the computation in each cycle. In the following discussion, we assume that only the processing unit can fail. The architecture of the processing cell (type-A) is shown in Fig. 24. Two horizontal and two vertical paths are allowed to traverse a cell at the same time. If there are more than one horizontal paths entering the cell, then the path associated with the local I/O port h1 precedes that associated with h2. Similarly, if there are two vertical paths entering the cell, then the path associated with v2 precedes that associated with v1. Also, there are three diagonal paths allowed to traverse a cell at the same time. If there are more than one diagonal paths entering the cell, then the path associated with d2 precedes that associated with d5, and d5 precedes d1.

4.4 Routing strategy

The cells in the array will route the data paths according to tables 2 to 15. It can be observed that the horizontal path H_j should meet the vertical path V_j and the diagonal path D_{m+j-i} at a common good cell. The routing strategy can be explained as follows:

1. The routing strategy of the horizontal and vertical paths is the same as suggested in [PaoD87]. A brief description of this strategy follows: The cells in the array route the data paths depending upon whether the cell is faulty or not. Basically, the routing strategy is to greedily extend the horizontal path toward the right boundary of the array. The horizontal path will make a turn and go one step downward if (a) it meets another horizontal path coming down from the cell above, or (b) it is moving together with a vertical path to find a good cell to perform the computation and it meets another vertical path coming from the cell above at a faulty cell. A vertical path V_j will be greedily extended downward to meet H_i. It will then move along with H_i to find an unused good cell to do the computation. Afterward, it will cross H_i and extend to H_{i+1},

and so on. V_j will take a step to the right if (a) it is moving along with a horizontal path to find an unused good cell, or (b) it meets another vertical path coming from the left.

2. If H_i meets D_k only, then H_i and D_k will move together either to the right or down (depending on the routing strategy of the horizontal path) till meeting the vertical path V_j and the computation is done.
3. If V_j meets D_k only, then V_j and D_k will move together either to the right or down (depending on the routing strategy of the vertical path) till meeting the horizontal path H_i and the computation is done.
4. When H_i meets V_j and D_k at a common good cell, the corresponding computation will be performed and the three paths H_i, V_j and D_k will be routed to the right, downward and diagonally, respectively. If other paths meet with the one above on a good cell, priority is for the three paths flowing all together.
5. When H_i meets V_j and D_k at a faulty cell, the three paths will be routed all together along H_i or V_j (depending on the routing strategy of the horizontal and vertical paths) to find a good cell to perform the required computation. Once the computation is performed the three paths H_i, V_j and D_k will be routed to the right, downward and diagonally, respectively.

4.5 Comments on special cases and tables

Going back to the type-A cell shown in Fig. 24, we can see the presence of seven inputs and seven outputs. All of the routing tables have been constructed under the assumption of the existence of spare cells that will enable to reconfigure an M x N array into m x n array (m ≤ M, n ≤ N). By grouping the horizontal and vertical inputs, we will be left with the three diagonal inputs, namely, d1, d2 and d5, respectively.

The diagonal inputs (dld2d5) can have eight possible combinations which are presented with both the horizontal and vertical inputs in tables 2 to 15. For every possible combination, two tables are needed, one if the cell is faulty and the other if the cell is good. The case of (000) represents 2-data flow systolic arrays which is already covered in [PaoD87] and presented in tables 16 and 17. Some of the entries of the tables will be explained below and the rest can be understood easily by following the same reasoning. For example, in table 10, consider the entries for (h1h2) as (01) and for (v1v2) as (01) which results in inputs and outputs as shown in Fig. 25. The computations will be performed between (h2v2d2). Once, the computation is performed, then h2 will flow to the right, v2 downward and d2 diagonally. The diagonal input d5 will flow horizontally to the right cell.The remaining entries of the different tables can be interpreted similarly.

Fig. 26 shows a reconfigured array. We will look at some interesting cases. For instance PE3 has (h2v2d2v1d5), the data routing will be done according to table 10 as follows: (h2v2d2) will be computed and h2 then flows to the right (PE4), v2 flows downward (PE8) and d2 flows diagonally (PE9). The other two inputs will flow to the right cell (PE4).

Now consider PE4 which has only three inputs (h2v2d2) and is faulty. It will be routed according to table 5. Then, the three inputs will pass unchanged to PE5. PE5 is also faulty and a boundary PE. Then, its three inputs will be redirected to PE10, according to the same table.

Finally, looking at PE12 which has (h1v1d1h2d2), the routing will be done according to table 8. The computations will be performed between (h1v1d1), then h1 flows to the right (PE13), v1 flows downward (PE17), d1 flows diagonally (PE18) and h2 and d2 will flow downward (PE17).

Fig. 27 illustrates the case of three diagonals (d1d2d5) reaching a cell. The inputs are routed according to tables 14 and 15.

Correctness of the reconfiguration follows from the following theorem and theorem 4.1.

Theorem 4.2

The routing tables satisfy the two invariants.

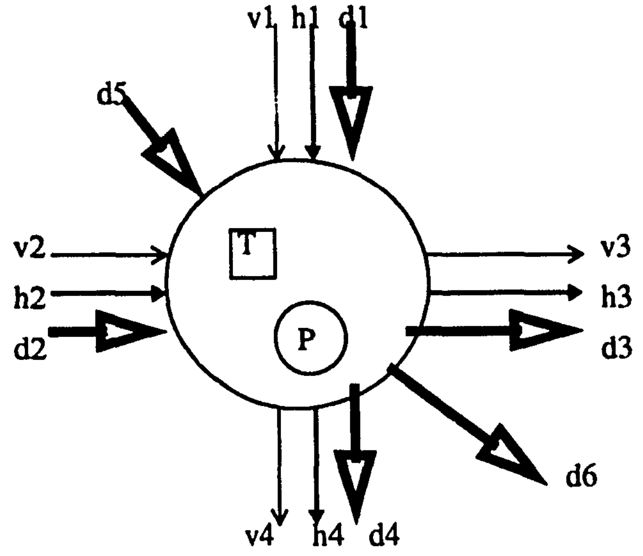
Proof

Tables 10 and 11 will be used to demonstrate the proof and similar proofs for other tables follow the same reasoning. If there are more than one diagonal (vertical) data received at the input ports of a cell, the diagonal path associated with d2 precedes that associated with d5 (the vertical path associated with v2 precedes that associated with v1). At the output ports, v3 goes to the right and v4 goes downward (e.g. entry h2v1v2d2d5). Also, d3 goes to the right, d4 goes downward and d6 goes diagonally. Hence, v4 precedes v3 and d4 or d6 precedes d3 at the output ports. Whenever there are two diagonal (one coming diagonally and the other coming from left) data entering a cell, d5 (the one coming diagonally) will be routed to d3 and d2 (the one coming from left) will be routed to d6 (if the cell is good and a computation was performed (e.g. entry h2v1v2d2d5). Otherwise, d2 will be routed to d4 (e.g. entry v1v2d2d5). Also, v1 will be routed to v3 and v2 routed to v4. Hence, the order of the paths are preserved when they pass through the cell and the first invariant is satisfied.

Consider the table for faulty cell (table 11). If originally h_i, v_j and d_k are to be processed at the faulty cell, then they will move together (in this case downward). Hence, the routing table 11 satisfies the second invariant.

For the good cell, the routing of data is the same as that of the faulty cell, except that the computation will be performed. If h_i, v_j and d_k have been processed, they will cross and h_i will move to the right, v_j downward and d_k diagonally (e.g. entry h2v1v2d2d5). Hence, routing table 10 satisfies the second invariant.

Q.E.D.



P: Processing Unit
T: Self-tester

Fig.24 Logical Structure of processing cell

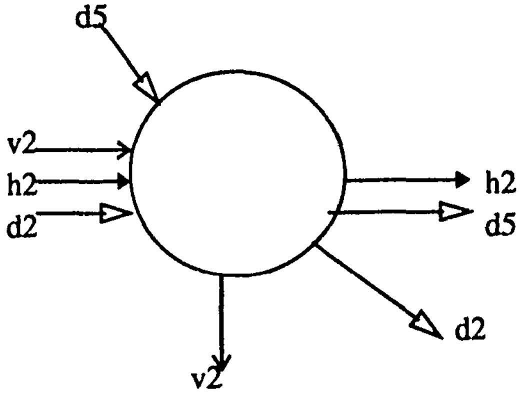


Fig. 25 Only (v2h2d2d5) available [entry from table 10]

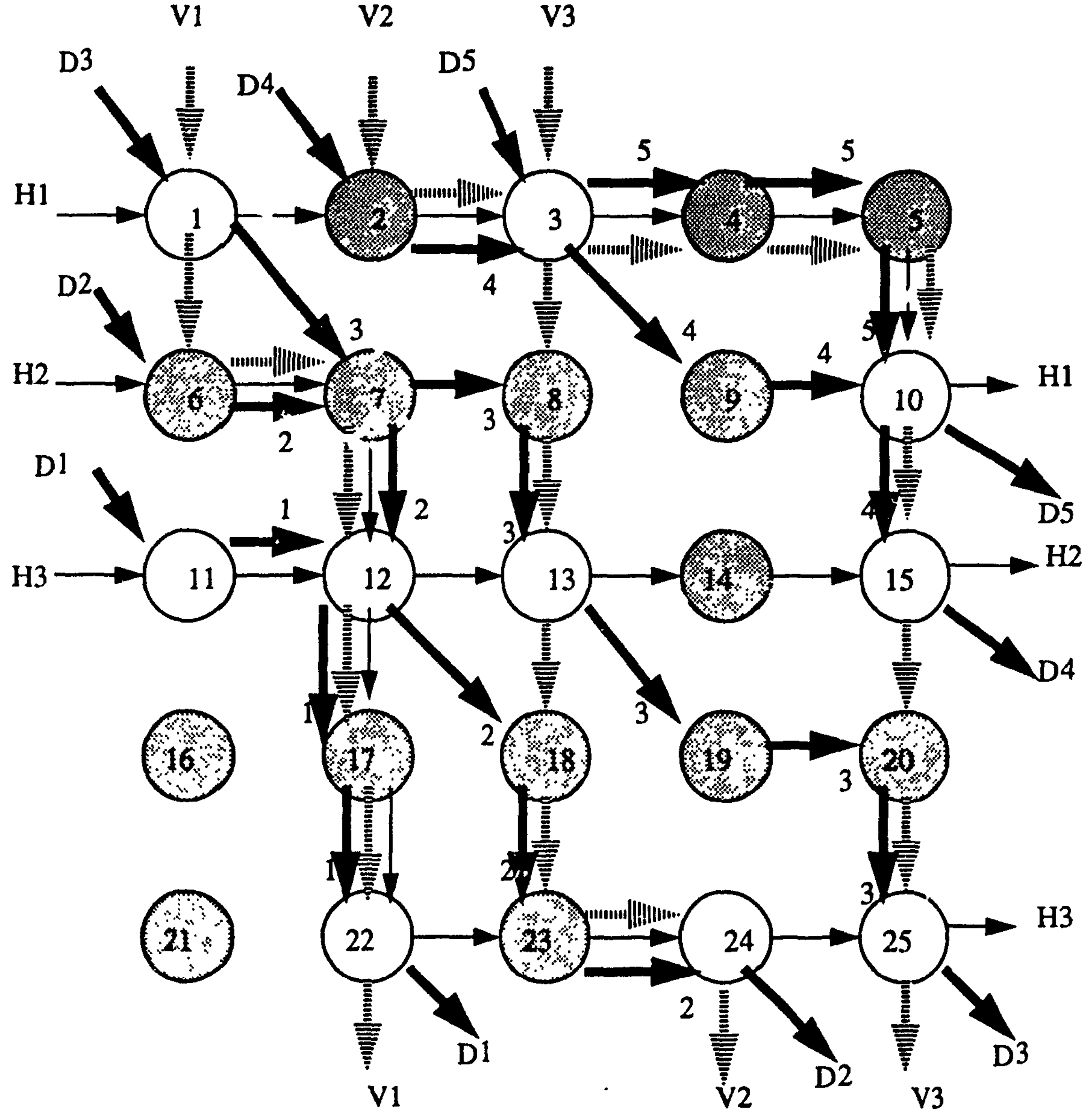


Fig. 26. A reconfigured array of type-A cells.

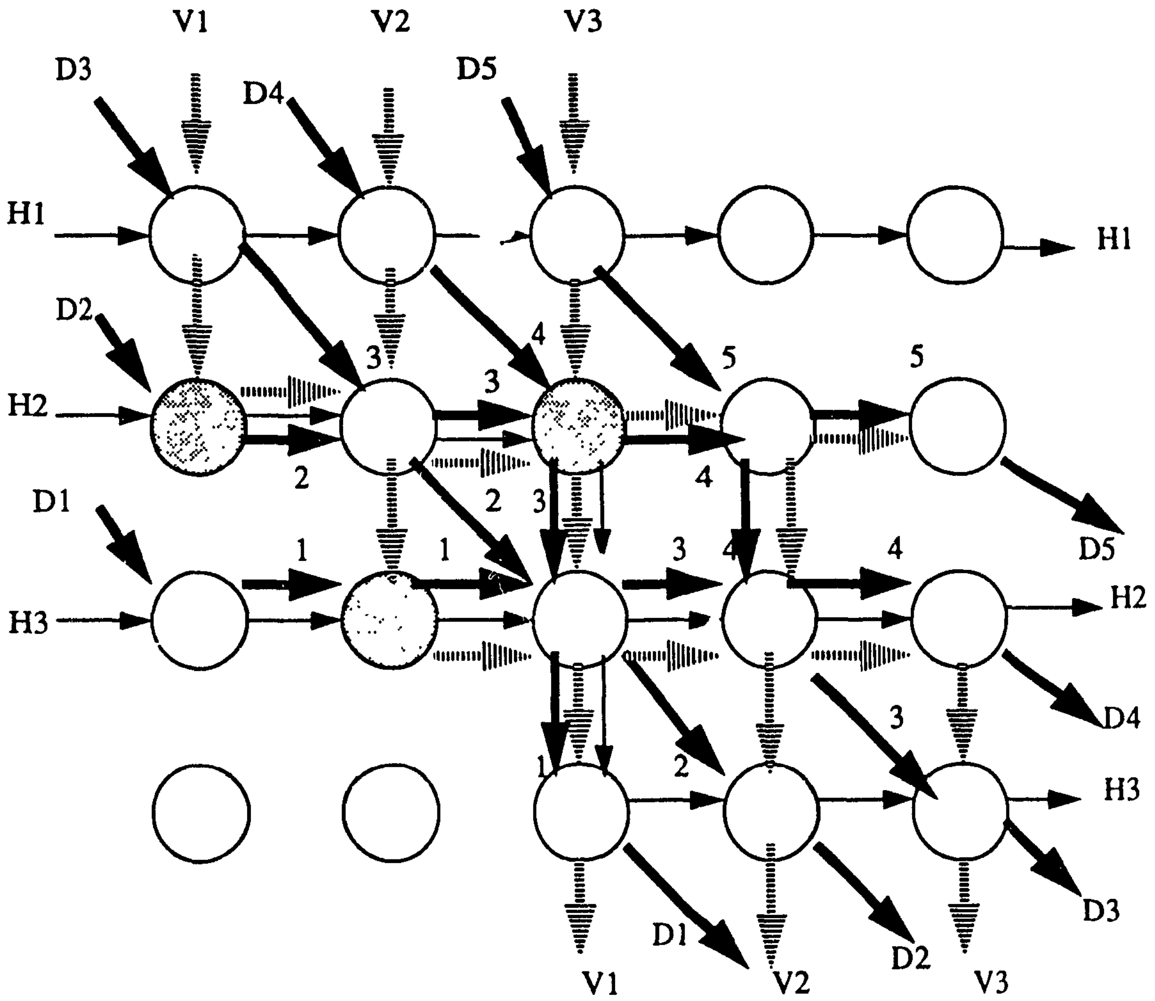


Fig. 27 A reconfigured array showing the case where three diagonals (d1d2d5) reach a cell.

The outputs specified in the box are:

v3	h3	d3	d6
v4	h4	d4	

In the routing tables of non faulty cells:

*: do the computation

-: cells at the right boundary.

In the routing tables of faulty cells:

+: cells at the right boundary

Empty slots in the tables mean that these combinations will not happen.

- Comments written on the top of a table for a good cell also apply to the corresponding table for a faulty cell.

- If h1 and h2 occur simultaneously, at least one of them should have the diagonal d1 or d2. As d1 and d2 are both absent, then this case can not happen. The same argument holds for (v1v2) entry in the table.

h1 h2 v1 v2	0 0				0 1				1 1				1 0			
0 0			d5			h2	d5							h1	d5	
0 1	v2		d5		v2*	h2*		d5*					v2*	h1*		d5*
1 1																
1 0	v1		d5		v1*	h2*		d5*					v1*	h1*		d5*

Table 2 Routing function of non faulty cell with only d5 present (case 001)

h1 h2 / v1 v2	0 0				0 1				1 1				1 0			
0 0			d5			h2	d5							h1	d5	
0 1	v2		d5		v2	h2	d5						v2	h1	d5	
1 1																
1 0	v1		d5		v1 v1*	h2 h2*	d5 d5*						v1	h1	d5	

Table 3 Routing function of faulty cell with only d5 present (case 001)

- When h1 happens, then it should be accompanied with d1 (h1d1) or with both v1 and d1 (h1v1d1). As d1 is absent, then this case can not happen and the entries where h1 appears are empty.

h1 h2 v1 v2	0 0				0 1				1 1				1 0			
0 0			d2	d2-		h2	d2	d2-								
0 1	v2		d2		v2*	h2 *		d2*								
1 1	v1 v2		d2		v1 v2*	h2*		d2*								
1 0	v1		d2		v1*	h2*		d2*								

Table 4 Routing function of non faulty cell with only d2 present (case 010)

h1 h2 / v1 v2	0 0				0 1				1 1				1 0			
0 0			d2	d2+		h2	d2	d2+								
0 1	v2		d2		v2 v2+	h2 h2+	d2 d2+									
1 1	v1 v2		d2		v1 v2	h2	d2									
1 0	v1		d2		v1 v1+	h2 h2+	d2 d2+									

Table 5 Routing function of faulty cell with only d2 present (case 010)

• When v2 happens, it should be accompanied with d2 (v2d2) or with both h2 and d2 (h2v2d2). As d2 is absent, then this case can not happen and the entries where v2 appears are empty.

v1 v2 \ h1 h2	0 0				0 1				1 1				1 0			
0 0						h2	d1			h1 h2	d1			h1	d1	
0 1																
1 1																
1 0	v1		d1		v1*	h2*		d1*	v1*	h1* h2		d1*	v1*	h1*		d1*

Table 6 Routing function of non faulty cell with only d1 present (case 100)

h1 h2 / v1 v2	0 0				0 1				1 1				1 0			
0 0						h2 h2+	d1 d1+			h1 h2	d1			h1	d1	
0 1																
1 1																
1 0	v1		d1		v1 v1+	h2 h2+	d1 d1+		v1	h1 h2	d1		v1	h1	d1	

Table 7 Routing function of faulty cell with only d1 present (case 100)

- When both d1 and d2 occur, this implies that at least one of the "h"s (h1 or h2) must occur, according to our routing strategy. Then, the case of h1 and h2 are both absent may not happen. Similarly, at least one of the "v"s should exist.

v1 v2 \ h1 h2	0 0				0 1				1 1				1 0			
0 0																
0 1					v2*	h2*	d1	d2*	v2	h1 h2	d1 d2					
1 1					v1 v2*	h2*	d1	d2*	v1* v2	h1* h2	d2	d1*	v1 v2	h1	d1 d2	
1 0									v1*	h1* h2	d2	d1*	v1*	h1*	d2	d1*

Table 8 Routing function of non faulty cell with both d1 and d2 present (case 110)

v1 v2 \ h1 h2	0 0				0 1				1 1				1 0			
0 0																
0 1					v2	h2	d1 d2		v2	h1 h2	d1 d2					
1 1					v1 v2	h2	d1 d2		v1 v2	h1 h2	d1 d2		v1 v2	h1	d1 d2	
1 0									v1	h1 h2	d1 d2		v1	h1	d1 d2	

Table 9 Routing function of faulty cell with both d1 and d2 present (case 110)

- If h1 happens, then it will be accompanied by d1. As d1 is absent, then the entries of h1 will be empty.

v1 v2 \ h1 h2	0 0				0 1				1 1				1 0			
0 0						h2	d5 d2									
0 1	v2		d5 d2		v2*	h2*	d5	d2*								
1 1	v1 v2		d5 d2		v1 v2*	h2*	d5	d2*								
1 0	v1		d5 d2		v1*	h2*	d5	d2*								

Table 10 Routing function of non faulty cell with both d2 and d5 present (case 011)

h1 h2 v1 v2	0 0				0 1				1 1				1 0			
0 0						h2	d5 d2									
0 1	v2		d5 d2		v2	h2	d5 d2									
1 1	v1 v2		d5 d2		v1 v2	h2	d5 d2									
1 0	v1		d5 d2		v1	h2	d5 d2									

Table 11 Routing function of faulty cell with both d2 and d5 present (case 011)

- If v2 happens, then it should be accompanied by d2. As d2 is absent, then the entries of v2 will be empty.

h1 h2 v1 v2	0 0				0 1				1 1				1 0			
0 0										h1 h2	d1 d5			h1	d1 d5	
0 1																
1 1																
1 0	v1		d1 d5		v1*	h2*	d5	d1*	v1*	h1* h2	d5	d1*	v1*	h1*	d5	d1*

Table 12 Routing function of non faulty cell with both d1 and d5 present (case 101)

v1 v2 \ h1 h2	0 0				0 1				1 1				1 0			
0 0										h1 h2	d1 d5			h1	d1 d5	
0 1																
1 !																
1 0	v1		d1 d5		v1	h2	d1 d5		v1	h1 h2	d1 d5		v1	h1	d1 d5	

Table 13 Routing function of faulty cell with both d1 and d5 present (case 101)

- When d1, d2 and d5 occur, this implies that at least one of the "h"s (h1 or h2) must occur and at least one of the "v"s should exist..

h1 h2 / v1 v2	0 0				0 1				1 1				1 0			
0 0																
0 1					v2	h2	d1 d2	d5	v2	h1 h2	d1 d2	d5	v2	h1	d1 d2	d5
1 1					v1 v2	h2	d1 d2	d5	v1 v2	h1 h2	d1 d2	d5	v1 v2	h1	d1 d2	d5
1 0					v1	h2	d1 d2	d5	v1	h1 h2	d1 d2	d5	v1	h1	d1 d2	d5

Table 14 Routing function of non faulty cell with d1, d2 and d5 present (case 111)

h1 h2 v1 v2	0 0				0 1				1 1				1 0			
0 0																
0 1					v2	h2	d1 d2	d5	v2	h1 h2	d1 d2	d5	v2	h1	d1 d2	d5
1 1					v1 v2	h2	d1 d2	d5	v1 v2	h1 h2	d1 d2	d5	v1 v2	h1	d1 d2	d5
1 0					v1	h2	d1 d2	d5	v1	h1 h2	d1 d2	d5	v1	h1	d1 d2	d5

Table 15 Routing function of faulty cell with d1, d2 and d5 present (case 111)

The outputs are specified in the square :

v3	h3
v4	h4

h1 v1 h2 v2	0 0	0 1	1 1	1 0
0 0		h2	h1 h2	h1
0 1	v2	h2+ v2 +	h1 v2+ h2+	h1 v2
1 1	v1 v2	C*	A*	v1+ h1+ v2
1 0	v1	h2* v1*	B*	h1+ v1+

*: do the computation.

+: do the computation if done flag = 0.

v1	h1
v2	h2

A: if f1 = 0 compute h1v1
else if f2 = 0 compute h2v2
else do not process

(f1 and f2 are flags accompanying the pair of data tokens on each side)

	h1
v1	h2

B: if f1 = 0 compute h1v1
else compute h2v1

v1	h2
v2	

C: if f2 = 0 compute h2v2
else compute h2v1

Table 16 Routing function of non-faulty cell (case 000) from [PaoD87].

The outputs are specified in the square :

v3	h3
v4	h4

v1 v2 \ h1 h2	0 0	0 1	1 1	1 0
0 0		h2	h1 h2	h1
0 1	v2	v2+ h2+ v2	h1 v2 h2	h1 v2
1 1	v1 v2	v1 h2 v2 h2+	v1 h1 v2 h2	v1 h1 v2
1 0	v1	v1 h2 v1* h2*	v1+ h1 v1 h2	v1+ h1 v1

+: if the incoming pair is not done

* : cells at the right boundary.

Table 17 Routing function of faulty cell (case 000) from [PaoD87].

CHAPTER 5

CONCLUSION

Techniques to achieve fault tolerance for three data flow systolic arrays both at run-time and at compile-time are presented. A run-time fault tolerance technique (3DFT) based on space-time mapping and hardware redundancy under the assumption that the latency of the computation is equal to or greater than three and the generated variables flow through the processing element has been proposed. The fault assumption is that only one PE of every three involved in doing the same computation can be faulty. The scheme can tolerate the occurrence of multiple faults (permanent and transient) under this condition. No assumption about any fault free elements or links is made. A processing element (PE) supporting the concept is proposed. The advantages of using 3DFT and the types of tolerated faults are discussed. The technique is not applicable in all cases; however, in the specific cases where it applies, fault tolerance can be achieved with a very low overhead.

Another run-time fault-tolerance scheme based on reconfiguration for unidirectional 3-data flow systolic arrays is also proposed. The distributed reconfiguration algorithm enables the cells of a processing array to dynamically restructure themselves based on local information. No a priori cell programming is required. A cell architecture is proposed and studied. The approach allows reconfiguration of the three data flow paths (horizontal, vertical and diagonal). Transient faults can be tolerated provided there are sufficient spare processors to replace the faulty processors.

For compile-time fault tolerance, an extension of the RCS-cut approach (static reconfiguration) is presented to cover the case of unidirectional 3-data flow systolic arrays. Also, comparisons with current reconfiguration techniques for hexagonal arrays is discussed showing the pros and cons of every scheme.

An interesting problem would be to simplify the cell architecture prop sed in the RCS-cut approach for the 3-data flow systolic arrays, as the current design needs a cell of nine inputs and nine outputs.

The distributed algorithms described in chapter 4 work for uni-directional 3-data flow systolic arrays. Extending the algorithms for bi-dirrctional data flow systolic arrays will be an interesting problem.

As for the two run-time approaches suggested, the 3DFT approach would be easy to implement and very efficient in the specific problems where it can be applied (matrix multiplication, transitive closure, etc.). While the dynamic reconfiguration approach is more general, it is more expensive in terms of hardware.

REFERENCES

Abra87 J.A. Abraham, P. Banerjee, C. Chen, W. Fuchs, S. Kuo and A. Reddy, "Fault tolerance techniques for systolic arrays", IEEE Computer, July 1987, pp 65-74.

Ball69 M. Ball, H. Hardie, "Majority Voter Design Consideration for TMR Computers", Computer Design, April 1969, pp 100-104.

Barb81 D.F. Barbe, "VHSIC Systems and Technology", IEEE Computer, Feb 1981, pp 13-22.

ChSW88 S. W. Chan, C. L. Wey, "The design of concurrent error diagnosable systolic arrays for band matrix multiplications", IEEE Trans. CAD Integr. Circuits and Syst, 7(1), 1988, pp 21 - 37.

Chen84 W-T. Cheng and J. Patel, "Concurrent Error Detection in Iterative Logic Arrays". Int. Symp. Fault Tolerant Computing, June 84, pp 10- 15.

ChHD85a H.D. Cheng, K.S. Fu, "Algorithm partition for a fixed-size VLSI architecture using space-time domain expansion", Int Symposium Circuits and Systems, 1985, pp 126-132.

Choi88 Y-H. Choi, M. Malek, "A Fault-Tolerant Systolic Sorter", IEEE Trans. on Computers, 37(5), 1988, pp 621- 624.

Cose88 R. J. Cosentino, "Concurrent error correction in systolic architectures", IEEE Trans. CAD Integr. Circuits and Syst, 7(1), 1988, pp 117- 125.

Fort85 J. A. B. Fortes, D. I. Moldovan, "Parallelism detection and algorithm transformation techniques useful for VLSI architectures design", Journal of Parallel and Distributed Computing, 1985, pp 277-301.

Fort85a J. A. B. Fortes, C. S. Raghavendra, "Gracefully degradable processor arrays", IEEE Trans. on Computers, 34(11), 1985, pp 1033-1044.

Fort86 J. Fortes, "Algorithm reconfiguration techniques for gracefully degradable processor arrays", Int. Workshop on Systolic Arrays (Systolic Arrays, Will Moore et al eds., Adam Hilger, 1987), pp 259-268.

Fuss84 D. Fussell, P. Varman, "Designing systolic algorithms for fault-tolerance", Proc. Int. Conf. Computer Design, 1984, pp 616-622.

Guib79 L.J. Guibas, H.T. Kung, C.D. Thompson, "Direct VLSI implementation of combinatorial algorithms", Caltech Conf. on VLSI, 1979, pp 509-525.

Gord84 D. Gordon, I. Koren and G. M. Silberman, "Embedding tree structures in VLSI hexagonal arrays", IEEE Trans. on Computers, 33(1), 1984, pp 104-107.

Gord87 D. Gordon, I. Koren and G. M. Silberman, "Restructuring Hexagonal Arrays of Processors in the presence of faults", Journal of VLSI and Computer Systems, vol. 2 no. 1- 2, 1987, pp 23-35

Gula86 R.K. Gulati and S. M. Reddy, "Concurrent error detection in VLSI array structures", Proc. Int. Conf. Computer Design, 1986, pp 488-491.

Hell85 D. Heller, "Partitioning big matrices for small systolic arrays", VLSI and Modern Signal Processing. S.Y. Kung (eds), Prentice-Hall, 1985, pp 764-767.

Huan84 K. H. Huang and J. A. Abraham, "Algorithm-based fault-tolerance for matrix operations", IEEE Trans. on Computers, 33(6), 1984, pp 518-528.

Hwan82 K.Hwang and Y. H. Cheng, "Partitioned matrix algorithms for VLSI arithmetic systems", IEEE Trans. on Computers, 31(12), 1982, pp 1215- 1224.

John89 B. Johnson, "Design and Analysis of Fault Tolerant Digital Systems", Addison-Wesley, 1989.

Jou86 J. Y, Jou and J. A. Abraham, "Fault-Tolerant matrix arithmetic and signal processing on highly concurrent computing structures", Proc. IEEE, 1986, pp 732-741.

Karp67 R.M. Karp et al, "The organization of computations for uniform recurrence equations", J. ACM, 14, 1967, pp 563-590.

KimJ85 J. H. Kim and S. M. Reddy, "A fault-tolerant systolic array design using TMR method", Proc. Int. Conf. Computer Design, 1985, pp 769-773.

KimJ87 J. H. Kim and S. M. Reddy, "On easily testable and reconfigurable two-dimensional systolic arrays", Proc. Int. Conf. Parallel Processing, 1987, pp 101-109.

KimJ87a J. H. Kim and S. M. Reddy, "Fault-Tolerant LU-Decomposition in a Two Dimensional Systolic Array", Concurrent Computations, Algorithms, Architecture and technology Edited by S. Tewksbury, B. Dickinson, S. Schwartz, 1987 Plenum Press, Chapter 29.

KimJ89 J. H. Kim and S. M. Reddy, "On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement". IEEE Transaction on Computers, 38(4), 1989, pp 515- 525.

Kore81 I. Koren, "A reconfigurable and fault-tolerant VLSI multiprocessor array", Proc of the Eighth Annual Symp. on Comp. Arch, 1981, pp425-441.

Kuma88 V. K. P. Kumar, Y.C. Tsai, "Mapping two dimensional systolic arrays to one dimensional arrays and applications", Proc. International Conference on parallel processing, 1988, pp 39-46.

Kuma89 V. K. P. Kumar, Y.C. Tsai, "On mapping algorithms to linear fault-tolerant systolic arrays", IEEE Trans. on Computers, 38(3), 1989, pp 470-478.

Kuma91 Sanjeev Kumar and Dharma P. Agrawal, "Design and analysis of highly reconfigurable hexagonal systolic arrays", Private Communication.

KuHT79 H. T. Kung, "Let's design algorithms for VLSI systems", Proc. Caltech Conf. on VLSI, 1979, pp 65-90.

KuHT82 H. T. Kung, "Why systolic architectures?", IEEE Computer, 15(1), 1982, pp 37-46.

KuHT84 H. T. Kung and M.S. Lam, "Wafer-scale integration and two level pipelining implementations of systolic arrays", Journal of Parallel and Distributed Computing, 1984, pp 32-63.

KuSY87 S. Y. Kung. VLSI array processors. Prentice-Hall, 1987.

KuSY87a S.Y. Kung, S.C. Lo, P.S. Lewis, "Optimal systolic design for the transitive closure and the shortest path problems", IEEE Transaction on Computers, 36(5), 1987, pp 603.

Kuo86 S. Y. Kuo and W. K. Fuchs, "Efficient spare allocation in reconfigurable arrays", Proc. Design Automation Conf., 1986, pp 385- 390.

Lala85 P. Lala, Fault Tolerant & Fault Testable Hardware Design. Prentice-Hall, 1985

LamC89 C. W. H. Lam, H. F. Li, R. Jayakumar, "A study of two approaches for reconfiguring fault-tolerant systolic arrays", IEEE Trans. on Computers, 38(6), 1989, pp 833.

Leig85 F. T. Leighton, C. E. Leiserson, "Wafer-scale integration of systolic arrays", IEEE Trans. on Computers, 34(5), 1985, pp 307-311

LiGJ85 G. J. Li, B. W. Wah, "The design of optimal systolic arrays", IEEE Trans. on Computers, 34(1), 1985, pp 66-77.

LiHF87 H. F. Li, D. Pao, and R. Jayakumar, "Dynamic Reconfiguration for fault-tolerant systolic arrays", Int. Conf. Parallel Processing, 1987, pp 110- 113.

LiHF89a H. F. Li, R. Jayakumar, C. Lam, Restructuring for fault-tolerant systolic arrays. IEEE Trans. on Computers, 38(2), 1989, pp 307-311.

LiHF89b H. F. Li, C. N. Zhang, R. Jayakumar, "Latency of computational data-flow and concurrent error detection in systolic arrays", Canadian Conf. VLSI, 1989, Vancouver, Oct 89, pp 251 -258

LiHF89c H. F. Li, D. Pao, and R. Jayakumar, "Systolic Arrays: Present State and Issues", Int Symp. on Computer Architecture and Digital Signal Processing, Hong Kong, Oct 1989, pp 69-74.

Maju90 A. Majumdar, C. S. Raghavendra, and M. A. Breuer, "Fault Tolerance in Linear Systolic Arrays using Time Redundancy", IEEE Trans. on Computers, 39(2), 1990, pp 269-276.

Mead80 C. Mead and L. Conway, Introduction to VLSI Systems. Addison-Wesley, Reading, MA, 1980.

Mira84 W. L. Miranker, "Space-time representations of computational structures", Computing, 32, 1984, pp 93-114.

Mold83 D. I. Moldovan, "On the design of algorithms for VLSI systolic arrays", Proc. IEEE, 71(1), 1983, pp 113-120.

Mold84 D. I. Moldovan, C. I. Wu, J. A. B. Fortes, "Mapping an arbitrarily large QR algorithm into fixed-size systolic array", Int. Conf. Parallel Processing, 1984, pp 365 - 373.

Mold85 D. I. Moldovan, "Tradeoffs between time and space characteristics in the design of systolic arrays", Proc. Int. Symp. Circuits and Systems, 1985, pp 1685-1688,

Mold85a D. I. Moldovan, J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays", IEEE Trans. on Computers, 35(1), 1985, pp 1- 12.

Mold87 D. I. Moldovan, "ADVIS: A software package for the design of systolic arrays", IEEE Trans. CAD Integr. Circuits and Systems, 6(1), 1987, pp 33-40.

Nava87 J. J. Navaro, J. M. Llaberia and M. Valero, "Partitioning: An essential step in mapping algorithms into systolic array processors", IEEE Computer, 20(7), 1987, pp 77-89.

Neli88 H. W. Nelis, E. F. Deprettere, "Automatic design and partitioning of systolic / wavefront array", Circuits, Systems and Signal Processing, 1988, 7(2), pp 235 - 252.

OKee86 M. T. O'Keefe, J. A. B. Fortes, "A comparative study of two systematic design methodologies for systolic arrays", Proc. Int. Conf. Parallel Processing, 1986, pp 672-675.

PaoD87 Derek Pao, "A Distributed Reconfiguration Approach for Transient Fault Tolerant Self-Reconfigurable Systolic Arrays and Performance Evaluations" Master thesis, 1987 Department of Computer Science, Concordia University.

Pate82 J. H. Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands", IEEE Trans. on Computers, 31(7), 1982, pp 589-595.

Piuri88 V. Piuri, "Fault-tolerant hexagonal arithmetic array processors", Microprocessing and Microprogramming 1988, vol 24, no. 1- 5, 1988, pp 629-636.

Prad86 D. K. Pradhan, Fault-Tolerant Computing Theory and Techniques, Vol 1 & 2, Prentice-Hall, 1986.

Quin84 P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations", Int. Symp. Computer Architecture, 1984, pp 208-214.

Renn84 D. A. Rennels, "Fault-Tolerant Computing -- Concepts and Examples", IEEE Trans. on Computers, 33(12), 1984, pp 1116-1129.

Rose83 A. L. Rosenberg, "The diogenes approach to testable fault-tolerant arrays of processors", IEEE Trans. on Computers, 32(10), 1983, pp 902-910.

Rote85 G. Rote, "A systolic array algorithm for the algebraic path problem (shortest path; matrix inversion)", Computing, vol 34, no 3, 1985, pp 191 - 219.

Rote86 G. Rote, "On the connection between hexagonal and unidirectional rectangular systolic arrays", Agean Workshop on Computing, 1986, pp 70-83.

Russ89 G. Russel and I. Sayers, Advanced Simulation and Test Methodologies for VLSI Design, Van Nostrand Reinhold (International), 1989.

Sami86 M. G. Sami, R. Stefanelli, "Reconfigurable architectures for VLSI processing arrays", Proc. IEEE, 74(5), 1986, pp 712-722.

Sami89 M. G. Sami, R. Stefanelli, R. Negrini, Fault-Tolerance through Reconfiguration of VLSI and WSI Arrays. MIT Press, 1989.

Shan88 W. Shang and J. A. B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies", International conference on systolic arrays, 1988, pp 393-402.

Shom87 L. A. Shombert, D. P. Siewiorek, "Using redundancy for concurrent testing and repairing of systolic arrays", Int. Conf Systolic Arrays, 1987, pp 393-402.

Siew82 D. P. Siewiorek and R.S Swaz, The theory and practice of reliable system design, Digital Press, 1982.

Varm86 P. J. Varman, I. V. Ramakrishnan, "Synthesis of an optimal family of matrix multiplication algorithms on linear arrays", IEEE Trans. on Computers, 35(11), 1986, pp 989-996.

Varm89 P. J. Varman, I. V. Ramakrishnan, "Optimal matrix multiplication on fault-tolerant VLSI arrays", IEEE Trans. on Computers, 38(2), pp 278-283. Intl. Symp. on Fault-Tolerant Computing, 1989, pp 244-249.

Wong85 Y. Wong, J. M. Delosme, "Optimal systolic implementations of N-dimensional recurrences", Int. Conf. Computer Design, 1985, pp 618-621.

Wong88 Y. Wong, J. M. Delosme, "Broadcast removal in systolic algorithms", Int. Conf Systolic Arrays, 1988, pp 403-412.

WuCC87 C-C. Wu, T-S. Wu, "Concurrent Error Correction in Unidirectional Linear Arithmetic Arrays", Proc. 17-th Intl. Symp. on Fault-Tolerant Computing, 1987, pp 136-141.

Zhan89 C. N. Zhang, H. F. Li, "Mapping data flow computation into universal systolic arrays", Technical Report, Department of Computer Science, Concordia University, 1989.