CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

A Distributed Reconfiguration Approach for Transient-Fault Tolerant Self-Reconfigurable Systolic Arrays and Performance Evaluations

Derek Chi Wai Pao

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
Montréal, Québec, Canada

July 1987

ISBN 0-315-37065-3

ABSTRACT

A Distributed Reconfiguration Approach for Transient-Fault Tolerant Self-Reconfigurable Systolic Arrays and Performance Evaluations

Derek Chi Wai Pao

A distributed reconfiguration approach based on local invariants technique is presented. The re-routing of data flow paths is based on local information only, and can be done without interfering with the computation. Hence, transient faults can be tolerated. Design of two self-reconfigurable uni-directional 2-data flow systolic arrays are presented. In the first design, both the horizontal and vertical paths are reconfigurable, whereas in the second design, only the vertical paths can be reconfigured. A distributed scheme to automatically generate an input schedule for the second design is presented. The internal configuration of the array can be made totally transparent to the host (user) by incorporating systolic routing networks made up of two types of simple routing cells. Performance of the two self-reconfigurable architectures are studied from both theoretical and practical point of views. It is shown that the two designs are exaetly $3n$-fault tolerant and $O(n^3)$

redundancy in the number of processing elements is sufficient to ensure successful recorfiguration. In the worst case, the first design requires m spare rows and one spare column to ensure tolerance of one transient fault, whereas only one spare column is required for the second design. Experimental performance evaluation of the distributed reconfiguration approach is studied via computer simulation. Simulation results reveal that (1) performance of the two proposed designs are comparable to that of the "static" counterpart, and (2) $O(n^2)$ redundancy in the number of processing elements is apparently sufficient to ensure successful reconfiguration. This reconfiguration approach is easily extenable to 3-data flow universal systolic arrays.

ACKNOWLEDGEMENT

I would like to express my sincere thanks to my supervisors, Dr. H. F. Li and Dr. R. Jayakumar for their guidance and encouragement throughout the course of this research.

Thanks also go to Dr. C. Lam and my colleagues for their encouragement and their useful comments on this research are greatly appreciated. Finally, but not least, I would like to thank Mr. Paul Gill, the system analyst, for his technical advice and support.

CONTENT

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Systolic array [18] is a desirable architecture for VLSI implementation because of its 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 of fault-tolerance in VLSI. The fault-tolerant scheme should not only improve the yield, but also ensure the 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.

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

There are two fault-tolerance approaches proposed in the literature, namely, fault-masking and reconfiguration.

1. Fault-masking

In this approach, extra components are used to mask off the effect of a fault instantaneously. A common technique employed is the Triple Modular Redundancy [5]. Three copies of the same computation are performed by three processors. Voting among the three results will determine the correct answer if only one of the three processors could fail. Direct implementation of this method often requires expensive duplications. Kim and Reddy [13] proposed a scheme that can reduce the number of processors required by a factor of two while tolerating only a limited classes of fault distributions. The design of a linear array to perform matrix ($A = a_{ij}$) vector (X) multiplication is given in this paper. The X data are duplicated over two consecutive time instances and the a_{ij} are 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.

2. Reconfiguration

This approach uses reconfigurable array structures made up of a 2-dimensional array of processing elements (cells) and an interconnection network that can be configured to embed the required processing topology (in this thesis we only deal with 2-D array). The processing array will first go through a testing phase to identify all the faulty cells. The array is then reconfigured to bypass all the faulty

cells. One property of systolic array is that the processing power (number of arithmetic/logic operations per unit time) of the system increases in proportion to the size of the array. Hence, the objective of the reconfiguration algorithm will be to optimize the utilization of the non-faulty cells. A graceful degradation of the performance of the system will result every time the faulty cells are excluded. Several reconfiguration algorithms have been proposed [7,15,19,22,34]. Leighton and Leiserson [22] proposed a reconfiguration algorithm for wafer scale integration of systolic array based on divide and conquer method. The interconnection wires are programmed by laser beam. In Rosenberg's method [34], 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. Fortes and Raghavendra [7] presented a reconfiguration algorithm based on row and column eliminations. The whole row or whole column in which a faulty cell resides is removed (bypassed) by setting the corresponding data switches. One drawback of these methods is that the interconnection wires may be lengthened, and hence the clock rate of the systolic array may be reduced.

Kung and Lam [19] proposed a systolic fault-tolerant scheme which maintains the original data flow pattern and the 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 a different delay). For any design, adding the same delay to all the edges in a cut and to those pointing from the host to the destination set of the cut will result in an equivalent design. 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.

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. The detection and correction of computation errors caused by operational faults (permanent

or transient) that arise during computation are left to the user. Abraham et al [10,12] proposed an *algorithm-based* fault-tolerant scheme for matrix operations that can detect and correct computation errors. The proposed scheme employs the row and column checksum technique. Given a vector (a_1, a_2, ..., a_n), two check elements WCS1 and WCS2 are appended to the vector where

$$WCS1 = \sum_{i=1}^{n} a_i$$

$$WCS2 = \sum_{i=1}^{n} a_i * 2^{i-1}$$

The algorithm is redesigned to operate on the encoded data and produce 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 that calculated from the output data indicates an error. The located error can be easily corrected based on the checksum produced by the algorithm. However, in general, the detection and correction of computation errors is by no means an easy job.

This thesis presents a distributed reconfiguration scheme that can tolerate permanent as well as transient faults. The proposed scheme follows Kung and Lam's 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 data tokens concerned 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 that, the systolic array should possess the following two capabilities.

1. Concurrent fault detection

   The processing cell should be able to tell concurrently with the normal operation whether the computation performed is correct or not.

2. Self-reconfiguration

   The systolic array should manage its own configuration without global knowledge of the fault distribution. Reconfiguration is done locally at the cells and can be changed at any time, subject to local status detected.

One problem due to the *dynamic* nature of the reconfiguration is that the I/O ports where data are input and output may change from time to time. To overcome this, a *universal* interface is proposed to make the configuration of the array transparent to the user. Data can be sent in and collected from fixed I/O ports at fixed times independent of the fault distribution.

The distributed reconfiguration algorithm will be demonstrated only for *2-dimensional 2-data flow* systolic arrays,

although it can be extended to universal systolic arrays. Two theoretical measures of performance, namely, a *lower bound* on the number of faults that can be tolerated and an *asymptotic redundancy* requirement to ensure successful reconfiguration of the reconfiguration algorithm are studied. Experimental performance evaluation is studied via computer simulation. Performance of the distributed approach is compared with that of two other static approaches, namely, the row-column elimination [7], and Kung and Lam's method [19], in which the reconfiguration is derived from global knowledge of the fault distribution.

Details of the distributed reconfiguration approach together with two example designs will be given in chapter 2. Chapter 3 will discuss the data collection and distribution networks. The transient-fault tolerant capability of the two designs described in chapter 2 will be discussed in chapter 4. Performance evaluations of the distributed reconfiguration approach can be found in chapter 5. Chapter 6 will highlight how the distributed reconfiguration approach can be extended to universal systolic arrays. Chapter 7 is the summary of the thesis.

Chapter 2

DISTRIBUTED RECONFIGURATION

Distributed reconfiguartion is characterized by the local decision making at individual processing cells. No global knowledge of the fault distribution is required in restructuring the array. The ideal goal is an autonomous subsystem that only requires some initialization. The user is free from any reconfiguration analysis. When viewed from outside, the array looks as if it is non-faulty. A reconfiguration is successful if an embedding of the required array is constructured in the faulty array; otherwise it is unsuccessful.

Koren [15] presented a distributed reconfiguration algorithm to restructure 2-D processor arrays. A reconfiguration command (the *header message*) is issued by the user which specifies the desired structure. A processing cell receiving a structuring message will transmit either another structuring message or an acknowledgement (a positive acknowledgement if the construction is completed or a negative acknowledgement 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 in this method.

1. Even though the reconfiguration algorithm is distributed, the 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.

In our 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 a novel *invariant technique*. A set of local invariants of the reconfiguration algorithm are identified. Intelligence is then incorporated into the processing cells to enforce these invariants.

Successful reconfiguration of an M x N array with faulty cells into a usable m x n ($m \leq M$, $n \leq N$) array of smaller size (fig. 1) is equivalent to the problem of determining a set of non-intersecting horizontal data paths H_1, H_2, ..., H_m and a set of non-intersecting vertical data paths V_1, V_2, ..., V_n such that each horizontal data path meets all the vertical data paths successively at good

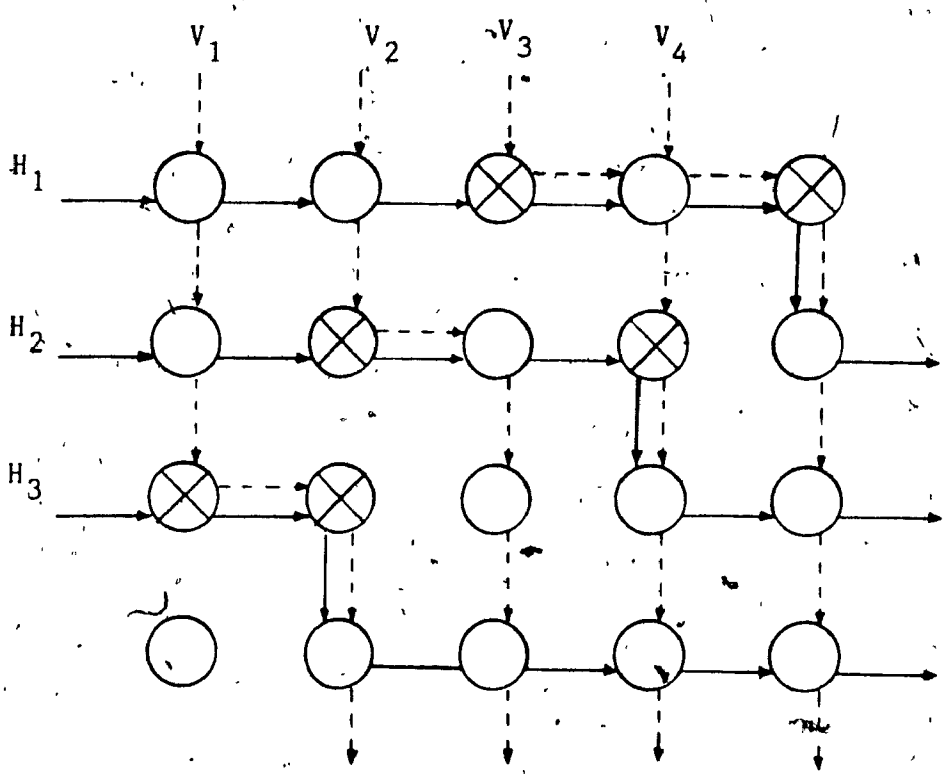


Fig. 1 A restructured 3x4 systolic array. Faulty cells are crossed.

(non-faulty) cells to perform the required computation, and vice versa. The horizontal paths are numbered from top to bottom and the vertical paths numbered from left to right. The horizontal (vertical) path H_i (V_i) precedes another horizontal (vertical) path H_j (V_j) for $i < j$. A path P_1 is said to have crossed another path P_2 if it is extended from one side of P_2 to the other side of P_2, and vice versa. The

word "path" and the phrase "data in the path" are used interchangably. Two invariants to ensure correct reconfiguration can be asserted:

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

Theorem 2.1

If an $M \times N$ array is reconfigured (with data re-routing) into a smaller array of size $m \times n$ in such a way that the two invariants are satisfied, and every horizontal path has crossed every vertical path, 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. When V_j crosses H_i, the required computation between them should have been completed (invariant 2). Hence, data on V_j will meet and be processed with data on the horizontal paths $H_1, H_2, \ldots, H_m$ successively in some common good cells. The same argument applies to the horizontal paths. **Q.E.D.**

2.1 Cell Architecture

Each cell has a processing unit together with a self-tester. The self-tester will check the result computed in every cycle. There are basically four different approaches to achieve concurrent fault detection. When choosing the technique to be used, we have to decide on the desired fault coverage and the hardware cost. A brief introduction to the four fault detection approaches is given below.

(a) Duplicate-and-compare

This straight-forward method involves duplicating the processing unit and comparing the results produced by each processing unit. A disagreement indicates an error. This simple method has a good coverage of faults, but it requires over 100% hardware redundancy.

(b) Coding [3,36]

In this approach, the input data are encoded and the processing unit is specially designed to produce encoded output. During normal (fault-free) operation, the encoded output only takes on a subset of all possible values (code-space). The appearance of a value that does not belong to the code-space indicates the occurence of fault(s). Many coding schemes have been proposed such as residue code, checksum code, and arithmetic code, etc. One drawback of this approach is that all the above mentioned coding schemes are based on the

stuck-at fault model. Recent studies on the failure modes of VLSI [4,28] reveal that many physical failures in VLSI cannot be modelled as stuck-at faults. Moreover, the hardware cost to implement such self-checking circuit [2,11] is relatively high.

(c) Recomputation with shifted operands [30]

This approach makes use of time redundancy. The normal computation is performed and the result is stored. The operands are then shifted left before the recomputation takes place. The result of the recomputation is then right shifted and compared with the previous result obtained. A mismatch indicates an error in the computation. The hardware requirement of this method is low but it involves 100% time redundancy.

(d) Algorithm-based fault detection

Local invariants of the specific algorithm are identified. Violation of one or more of these invariants at a cell indicates an error. The advantage of this high level fault detection method is that little hardware/time overhead is required. Choi and Malek proposed a fault-tolerant systolic sorter [6] which employs this fault detection approach. The sorting algorithm is as follows. A linear array of N cells is used. Each cell[i] has two registers, P[i] and Q[i]. The sorting is done by the following two phase operation:

```
Procedure Sort
  BEGIN
     initialize P[i], Q[i] to MAX;
     FOR i=1 TO N/2 DO (N cycles)
        PARBEGIN          (*input phase*)
          Q[i] ← Q[i-1];
          IF P[i] > Q[i] THEN swap(P[i],Q[i]);
        PAREND
     FOR i=1 TO N/2 DO (N cycles)
        PARBEGIN          (*output phase*)
          P[i-1] ← P[i]; (P[N/2] ← MAX)
          IF P[i] > Q[i] THEN swap(P[i],Q[i]);
        PAREND
  END
```

The proposed method is based on the two invariants: (1) for all i's the input keys and the output keys of cell[i] must have the same checksum; and (2) for all i's, cell[i] in the array pops off completely sorted ouput for a correct recursion. The linear array is partitioned into *b* blocks with the first cell of each block operating in test mode. Data are duplicated at the test cells using bypass links. A test cell will check the monotone property of the data stream, and computes the checksums of the input keys and that of the output keys. A mismatch between the two checksums indicates the existence of a fault within that block. One diffi-

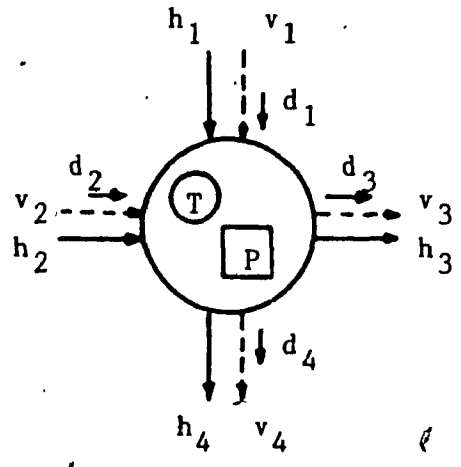
culty in designing algorithm-based fault detection schemes is that the relationships between the functional faults and the faults in the physical layer have not been well understood. Further research in this area is expected.

The fault detection technique employed will be application specific and the detail of which is beyond the scope of this thesis. Hence, in this thesis the phrase "cell architecture" refers to the organization of the interconnection links and the data switches only. In the following discussion, it is assumed that only the processing unit may fail. The interconnection wires, data switches, and self-teser will not fail. This assumption is justified as the data switches have a smaller active area than the processing unit and the fabrication process of the interconnection wires is simpler. The self-tester, depending on the implementation, can be made fail-safe.

The cell architecture and the routing invariants adopted are interdependent. A tradeoff between the flexibility and simplicity has to be made when designing the routing function of the cell. Two cell architectures, namely, type-A and type-B, are proposed and studied in depth. The type-B cell is less flexible than the type-A cell, but it has several advantages over the type-A counterpart, as shown later.

2.2 Design-1 (DR1)

Type-A cells (fig. 2) are used in this design. Two horizontal and two vertical paths are allowed to traverse a cell at the same time. If there is more than one horizontal path entering the cell, then the path associated with the local I/O port h_1 precedes that associated with h_2. Similarly, if there are two vertical paths entering the cell, then the path associated with v_2 precedes that associated with v_1. A single-bit logic flag d_i (done flag) accompanying the pair of data tokens on each side is used to indicate whether or not the computation between that pair of data has been completed.



T : Self-tester

P : Processing unit

Fig. 2 Type-A cell architecture

2.2.1 Routing Strategy

The cells in the array route the data paths according to tables 1 and 2 depending upon whether the cell is fault-free or not. Basically, the routing strategy is to greedily extend the horizontal path toward the right boundary of the array. To preserve the two invariants, the horizontal path will make a turn and go one step downward if

1. it meets another horizontal path coming down from the cell above, or
2. it is moving together with a vertical path to find a good (fault-free) cell to perform the computation, and 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

1. it is moving along with a horizontal path to find an unused good cell, or
2. it meets another vertical path coming from the left.

Referring to the routing tables 1 and 2, $Ih_i/Iv_j = 1$ if there is valid data present at the local input port h_i/v_j. Some of the entries of the tables will be explained below and the rest can be understood easily by following the same

The outputs are specified in the square :

v_3	h_3
v_4	h_4

Iv_1 Iv_2 \ Ih_1 Ih_2	0 0	0 1	1 1	1 0
0 0		h_2	h_1 h_2	h_1
0 1	v_2	h_2^+ v_2^+	h_1 v_2^+ h_2^+	h_1 v_2
1 1	v_1 v_2	C*	A*	v_1^+ h_1^+ v_2
1 0	v_1	h_2^* v_1^*	B*	h_1^+ v_1^+

* : do the computation
\+ : do the computation if done flag = 0

A : if $d_1 = 0$ compute h_1v_1

else if $d_2 = 0$ compute h_2v_2

else do not process

v_1	h_1
v_2	h_2

B : if $d_1 = 0$ compute h_1v_1

else compute h_2v_1

	h_1
v_1	h_2

C : if $d_2 = 0$ compute h_2v_2

else compute h_2v_1

v_1	h_2
v_2	

Table 1 Routing function of non faulty type-A cell

The outputs are specified in the square :

v_3	h_3
v_4	h_4

Iv_1 Iv_2 \ Ih_1 Ih_2	0 0	0 1	1 1	1 0
0 0		h_2	h_1 h_2	h_1
0 1	v_2	v_2^+ h_2 v_2	h_1 v_2 h_2	h_1 v_2
1 1	v_1 v_2	v_1 h_2 v_2 h_2^+	v_1 h_1 v_2 h_2	v_1 h_1 v_2
1 0	v_1	v_1 h_2 v_1^* h_2^*	v_1^+ h_1 v_1 h_2	v_1^+ h_1 v_1

+ : if the incoming pair is not done

* : cells at the right bonndary

Table 2 Routing function of faulty type-A cell

reasoning.

1. Referring to fig. 3a, the cell will perform the computation between h_2 and v_1. If the result obtained is correct, then the two data paths will cross each other as shown in fig. 3b. If the cell is faulty, then the computed result will be discarded. The two data paths will then travel in parallel and the done flag will be set to '0' (computation not done) as in fig. 3c.

2. Referring to fig. 4, the horizontal path at h_1 must have met and crossed the vertical path v_2 before. Hence, no computation will be performed.

3. There are two horizontal paths entering the cell (fig. 5). Since h_1 preceds h_2, h_1v_1 should be processed before h_2v_1. Therefore, if $d_1 = 0$, the processor will compute h_1v_1, otherwise, it will compute h_2v_1. If the cell is faulty and $d_1 = 0$, then v_1 will move along with h_1, otherwise, it will cross h_1 and travel in parallel with h_2.

4. There are four data paths entering the cell (fig. 6). In order to preserve the invariants, the routing of the data paths are fixed. If $d_1 = 0$ then h_1v_1 will be computed, otherwise, if $d_2 = 0$ then h_2v_2 will be computed. If both d_1 and d_2 are '1', no computation can be performed. The routing of a faulty cell is exactly the same as that of a good cell, except that no useful com-

putation will be performed.

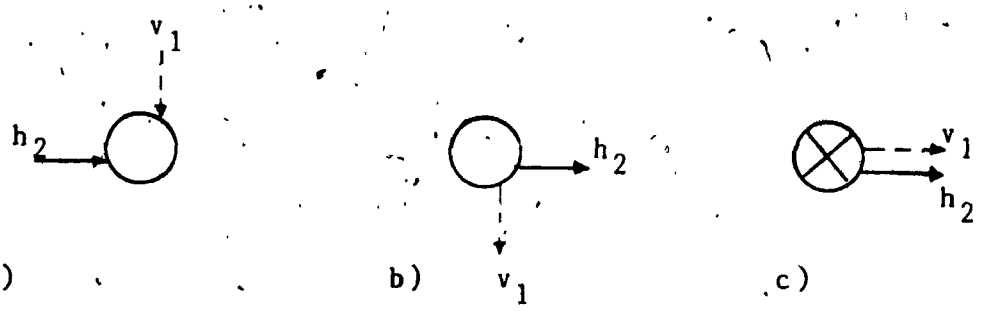


Fig. 3 Routing example 1. a) input data paths; b) routing of good cell; c) routing of faulty cell.

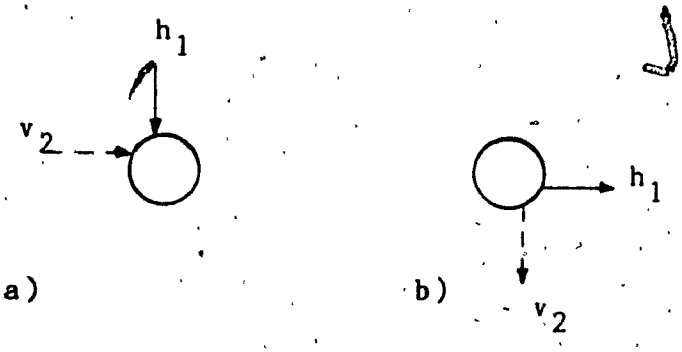


Fig. 4 Routing example 2. a) input data paths b) routing of good and faulty cell

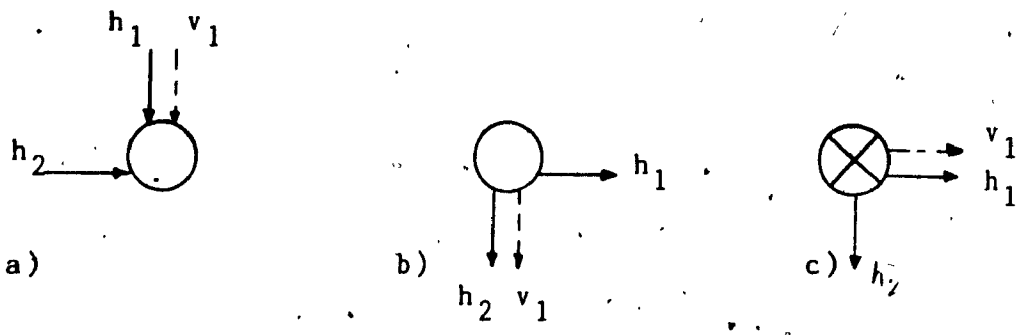


Fig. 5 Routing example 3. a) input data paths; b) routing of good cell or input done flag = 1; c) routing of faulty cell and input done flag = 0

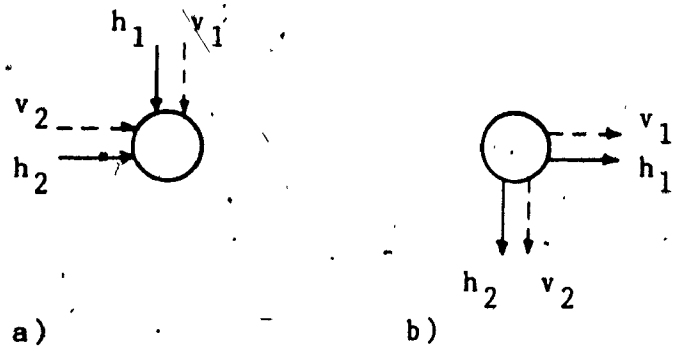


Fig. 6 Routing example 4. a) input data paths b) routing of good and faulty cells

Theorem 2.2a

The routing tables 1 and 2 satisfy the first invariant.

Proof

An abstracted proof by induction on the cell position is presented. If there is more than one horizontal (vertical) data received at the input ports of a cell, the induction hypothesis asserts that the horizontal path associated with h_1 precedes that with h_2 (the vertical path associated with v_2 precedes that with v_1). At the output ports, h_3 (v_3) goes to the right and h_4 (v_4) goes downward. Hence, h_3 (v_4) precedes h_4 (v_3) at the output ports. Whenever there are two horizontal (vertical) data entering a cell (faulty or not), h_1 (v_1) will be routed to h_3 (v_3) and h_2 (v_2) routed to h_4 (v_4). Hence, the order of the paths are preserved when they pass through the cell. **Q.E.D.**

Theorem 2.2b

The routing tables 1 and 2 satisfy the second invariant.

Proof

Consider the table for faulty cell. An incoming pair of paths will be crossed if and only if the corresponding done flag is set to '1'; otherwise, they will be kept in pair and the done flag copied to the output ports. If originally h_i and v_j are to be processed at the faulty cell, then they will be paired and the corresponding done flag is set to '0'. Hence, the routing table 2 satisfies the second

invariant.

For the good cell, the routing of an incoming pair of data is the same as that of the faulty cell, except that the computation will be performed if the condition allows. If h_i and v_j have been processed, then they will cross each other or be kept in pair and the done flag is set to '1'. Hence, the routing table 1 satisfies the second invariant. **Q.E.D.**

Theorem 2.3

Horizontal path H_i and vertical path V_j will be processed exactly once if the reconfiguration is successful.

Proof

The first time H_i meets V_j, H_i should be moving horizontally, and V_j moving vertically. After they have been processed, they will eventually cross each other. If H_i were to meet V_j the second time, H_i should be moving downward and V_j moving to the right. For all the four possible cases shown in tables 1 and 2, the two paths are simply separated without doing any computation. **Q.E.D.**

Fig. 7 depicts an example reconfigured array of type-A cells. It is obvious that the reconfigurability of a faulty array depends on when and where data are sent into the array (*input schedule*). It is observed that in the DR1 design, the route of a horizontal path depends on the routes of the vertical paths, and vice versa. Hence, it is impossible to

derive a good input schedule without global knowledge of the fault distribution. This is to say that to obtain an optimally reconfigured array, some kind of *offline analysis* is required. However, this is not desirable for the distributed reconfiguration approach. If no offline analysis is allowed, a simple input schedule that packs all the inputs towards the top-left corner can be adopted. One drawback of this strategy is that the reconfigurability of the array will be sensitive to the faults in the top-left quadrant. In the next section, we discuss the second design which overcomes this drawback. For this design a good input schedule can be derived automatically in approximately linear time.

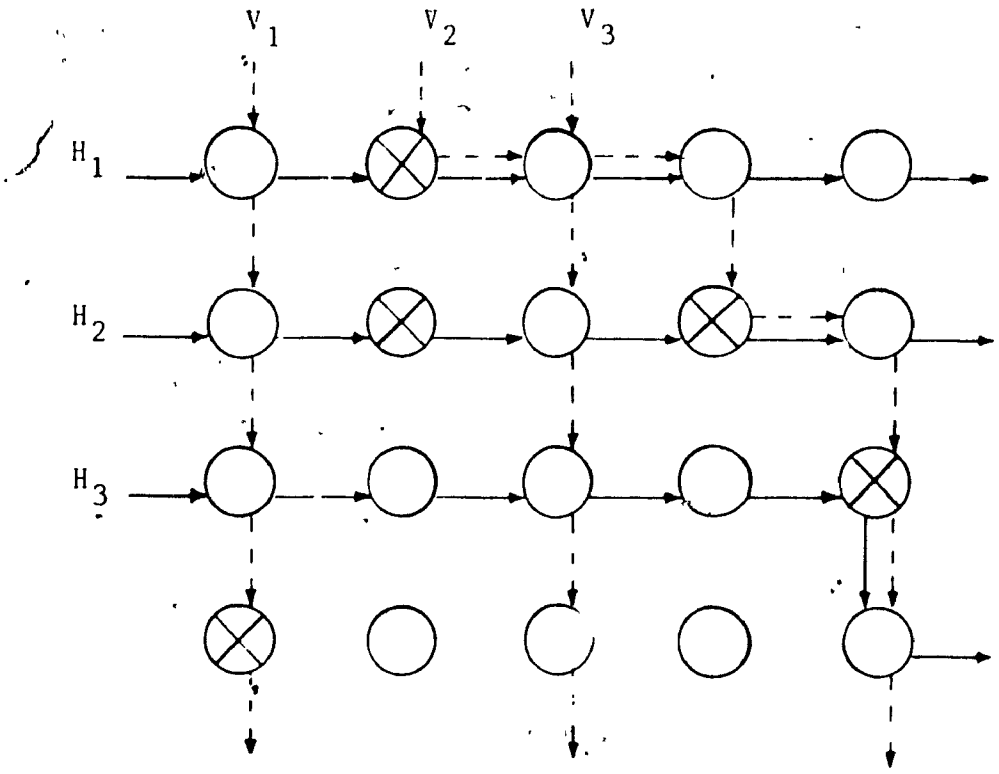
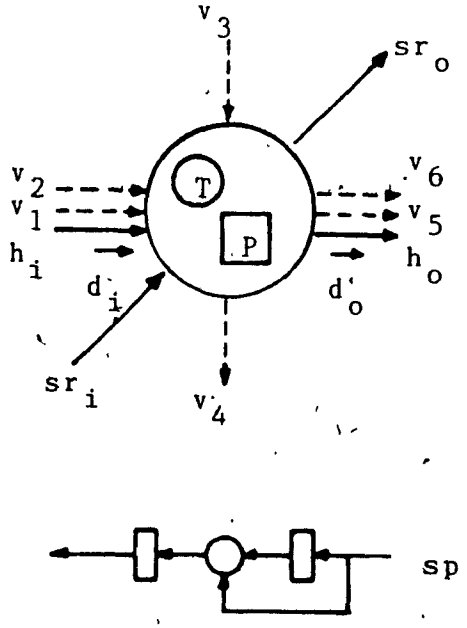


Fig. 7 A restructured 3x3 array of type-A cells

2.3 · Design-2 (DR2)

Type-B cells (fig. 8) are used in this design. For the type-B cells the routing of horizontal paths is fixed (formed by the horizontal rows), but the vertical paths are reconfigurable. Up to three vertical paths can traverse the same cell simultaneously. However, the input port v_2 can have valid data only if there is valid data at v_1. By confining the routing of the horizontal paths, the interdependence between the horizontal and vertical paths can be eliminated. Heuristics can be applied to select the horizontal rows (horizontal paths), and generate the input schedule vertical data without knowing the fault distribution.



T : Self-tester
P : Processing unit

Fig. 8 Type-B cell architecture

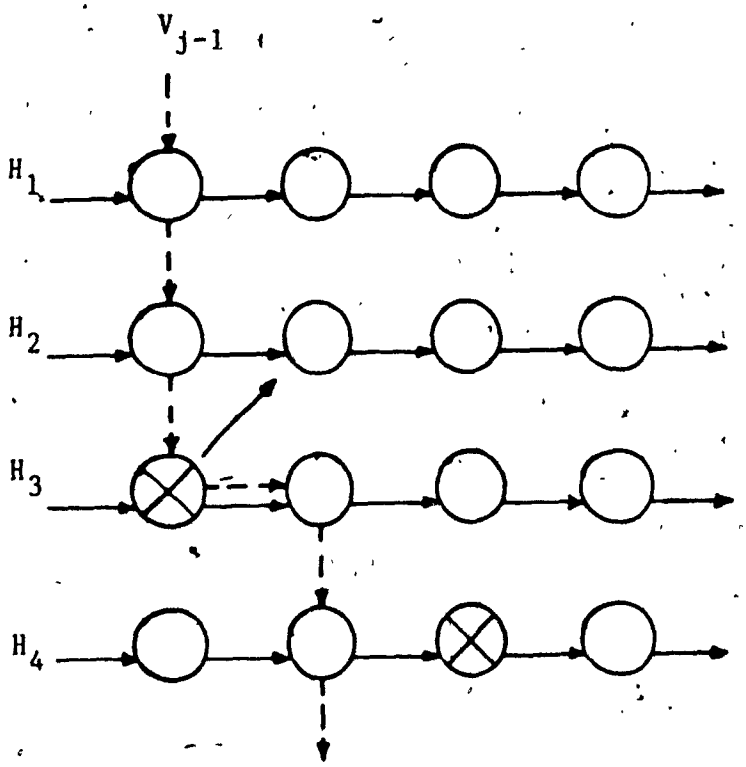
Two done flags, namely, d_i and d_o, are used to indicate whether the computation between h_iv_1 and h_ov_5, respectively, have been completed or not. The sr_o (send-to-right) signal can be set to request the cell on the upper right not to send vertical data downward in the next cycle. Consequently, on receiving the sri request, a cell will avoid sending vertical data downward and forward them horizontally instead. The sr_o signal in a cell will be set if

1. the cell is faulty in the current cycle and useful computation should have been performed if it were fault-free, or
2. it receives the sr_i request from the cell on its bottom left and one or more vertical data from its neighbors.

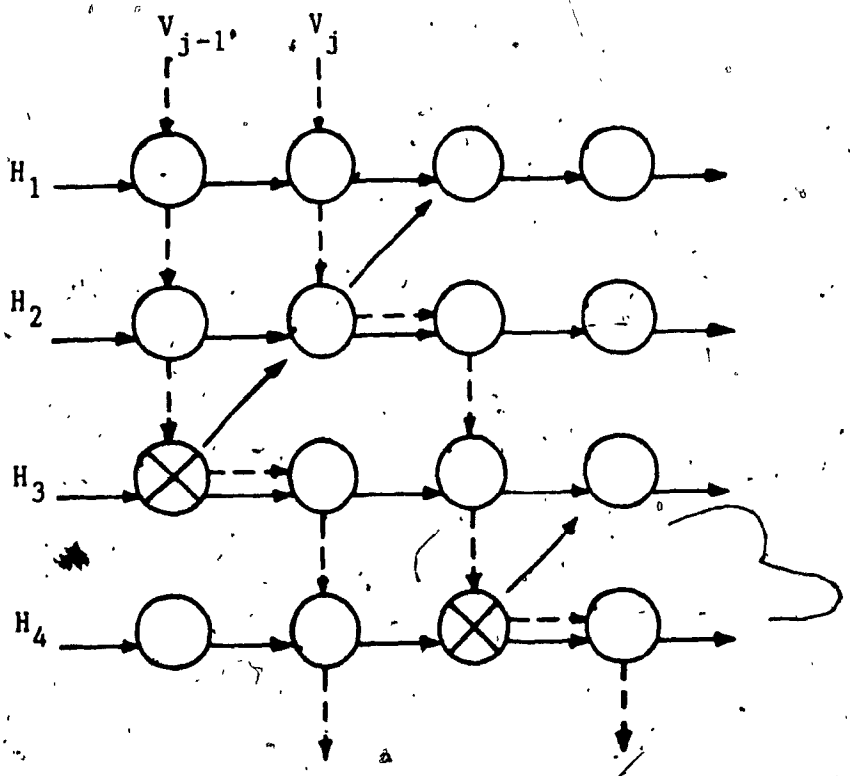
The sp (self-programming) signal is used in the initialization phase for horizontal row selection which will be elaborated in section 2.3.2.

2.3.1 Routing strategy

Given the set of horizontal paths, H_1, H_2, ..., H_m, an optimal set of non-touching vertical paths is determined using a "*greedy*" approach (fig. 9). The reason why non-touching vertical paths are desirable is that if a transient fault or a new fault arises, there will be enough room to ensure successful local re-routing of data dynamically. Suppose the vertical paths V_1, V_2, ..., V_{j-1} have been constructed. V_j will be constructed starting from the first



a) path V_j to be constructured



b) path V_j is greedily extended downward

Fig. 9 Greedy approach for vertical path generation

unused (not included in V_{j-1}) cell on the first row. Local re-routing extends V_j greedily downward. Whenever V_j sees the sr_i request (the cell below is occupied), it will take one step to the right and the corresponding processing cell will set its sr_o signal. When V_j meets H_1, it moves along with H_1 to find a good cell to perform the computation. V_j will then cross H_1 and extend to H_2. This process will be repeated until it crosses H_m. Each processing cell that receives horizontal data routes the vertical data inputs according to tables 3 and 4 depending upon whether it is good or faulty. A processing cell (faulty or not) that does not receive horizontal data will perform exactly the same routing function as that of a good cell except that no computation will be performed. Note that the sr_o signal is not set even if vertical data is sent to the right by a good cell (table 3) if the sr_i input of the cell is not set. These cases only arise during the dynamic re-routing of data when a new fault is detected during computation. Such cases only last for one cycle, and therefore the sr_o signal should not be set, otherwise the data tokens might have taken unnecessary right steps.

Theorem 2.4a

Routing tables 3 and 4 satisfy the first invariant.

Proof

An abstracted proof by induction on the cell position is presented. The vertical paths enter the cell at the local

The outputs are specified in the square :

sr_o v_6	
v_4	v_5

sr_i \ Iv_1 Iv_2 Iv_3	0 0	0 1	1 1	1 0
0 0	0		0 A* v_1 v_2	0 v_1^+
0 1	0 v_3^*		0 v_3 A* v_1 v_2	0 B* v_1 v_3
1 1	1 v_3^*		1 v_3 A* v_1 v_2	1 v_3 v_1^+
1 0	0		1 v_2 v_1^+	1 v_1^+

* : do the computation
\+ : do the computation if $d_i = 0$

A : compute $h_i v_1$ if $d_i = 0$
else compute $h_i v_2$

B : compute $h_i v_1$ if $d_i = 0$
else compute $h_i v_3$

Table 3 Routing function of non-faulty type-B cell

The outputs are specified in the square :

sr_o	v_6
v_4	v_5

sr_i \ Iv_1 Iv_2 Iv_3	0 0	0 1	1 1	1 0
0 0	0		1 A* v_1 v_2	0 B* v_1
0 1	1 v_3		1 F* v_3 v_1 v_2	1 C* v_1 v_3
1 1	1 v_3		1 F* v_3 v_1 v_2	1 v_3 v_1
1 0	0		1 v_2 v_1	1 v_1

A : if $d_i = 0$

1	v_2
	v_1

B : if $d_i = 0$

1
v_1

C : if $d_i = 0$

1	v_3
	v_1

F : reconfiguration fails if $d_i = 0$

Table 4 Routing function of faulty type-B cell

input ports v_1, v_2, v_3 and leave the cell at output ports v_4, v_5 and v_6 of the cell. If there is more than one vertical data received at the input ports of a cell, the induction hypothesis asserts that the path associated with v_i precedes that associated with v_j, for $i < j$ ($i, j = 1,2,3$). At the output ports, v_4 goes downward whereas v_5 and v_6 are connected to the first and second input ports (v_1 and v_2), respectively, of the cell on the right. At the output ports of the cell, the re-routed data at v_p precedes that at v_q for $p < q$ ($p, q = 4,5,6$). If there are two or more vertical data entering a cell, then for any two of the vertical data, say v_i and v_j, v_i will be routed to v_p and v_j routed to v_q. The two routing tables 3 and 4 ensure that if $i < j$ then $p < q$. Hence the order of the vertical paths is preserved when they pass through the cell. **Q.E.D.**

Theorem 2.4b

Routing tables 3 and 4 satisfy the second invariant.

Proof

Consider the routing table of a type-B faulty cell. Excluding the failure cases, the vertical data associated with port v_1 that enters the cell will cross the horizontal path only if the input done flag d_i is equal to 1 (i.e. the computation was already performed). For the case of a good cell, the crossing of a vertical path with the horizontal path occurs only if the corresponding computation has been

performed at that cell or previously. **Q.E.D.**

Theorem 2.5

If the external input ports of the vertical paths to the array satisfy the condition that the vertical path V_j is being sent into the array at the first unused (not included in V_{j-1}) cell in the top row to the right of that for V_{j-1}, then a maximum set of non-touching vertical paths can be obtained.

Proof

Suppose j-1 non-touching vertical paths that are optimally packed to the left have been constructed. V_j is then sent into the array from the first unused cell to the right of V_{j-1}. V_j is then greedily extended downward. The greediness assures that no other vertical path V'_j can be constructed to the left of V_j without touching V_{j-1} since V_j will only take a step to the right at a faulty cell or when it sees the sr_i request. Hence, if every cells on the top row are tried out, the set of non-touching vertical paths obtained is optimal. **Q.E.D.**

Fig. 10 depicts an example reconfigured array of type-B cells. The method employed to automatically generate the input schedule is presented in the next section.

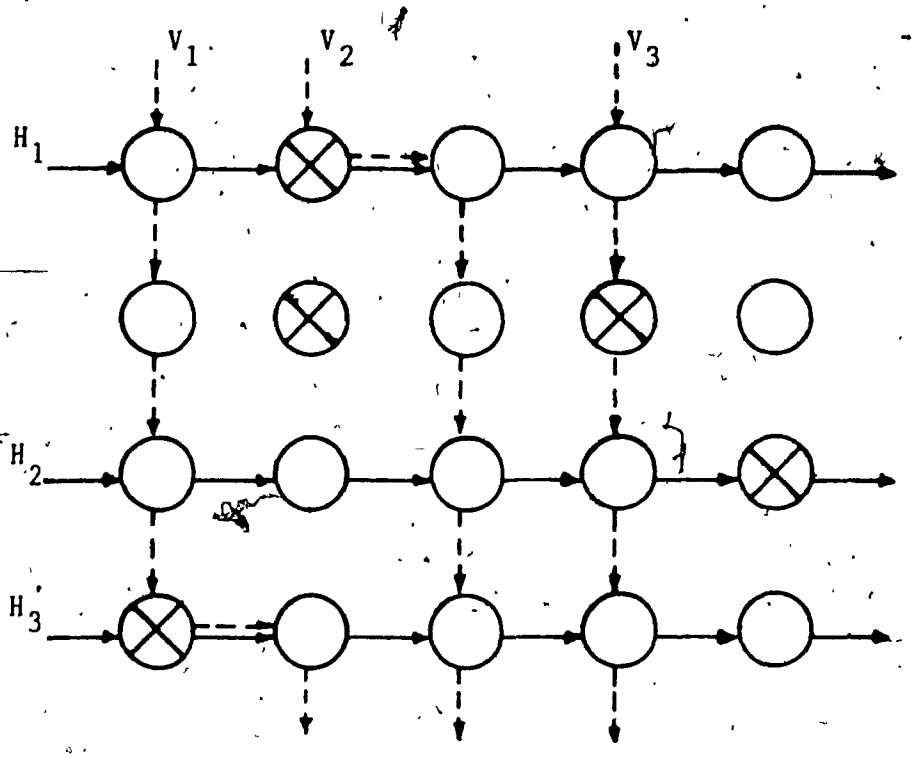


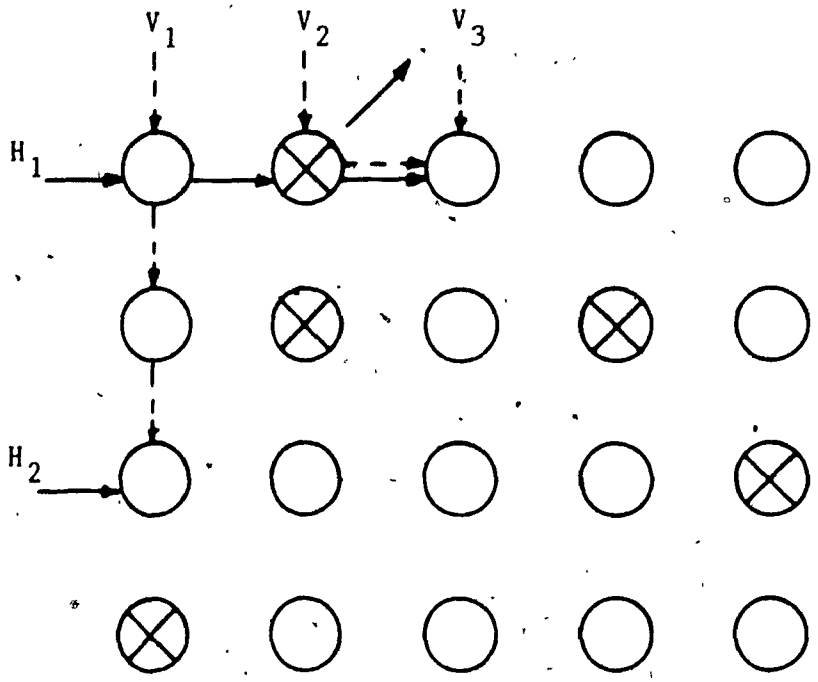
Fig. 10 A restructured 3x3 array of type-B cells

2.3.2 Input Schedule Generation

Finding the optimal reconfiguration of a faulty array is a combinatorial problem. The reconfigurability of the array depends very much on the input schedule. In the distributed reconfiguration approach, the fault distribution is not known and enumeration of all the possible input schedules is not practical either. Heuristics are applied to find a good

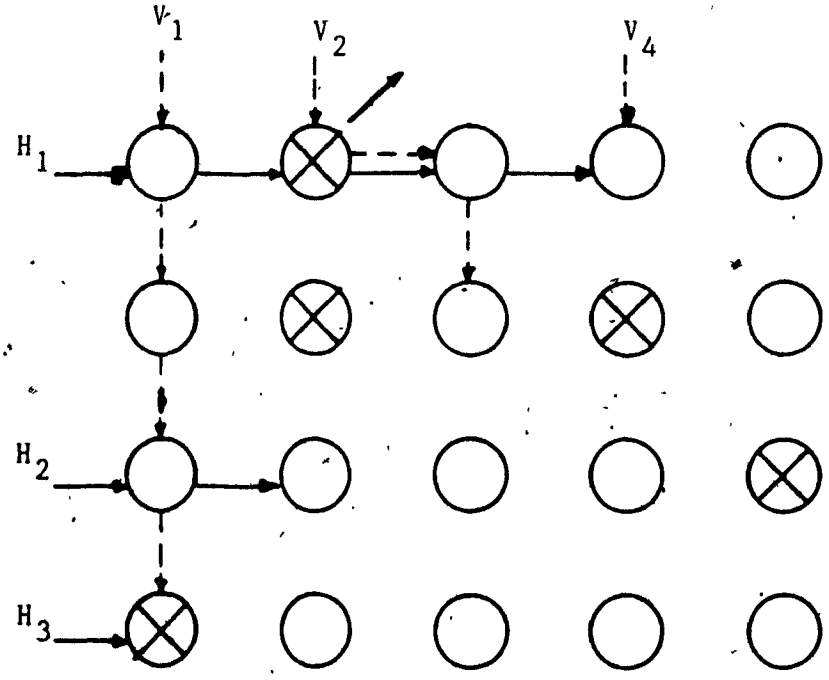
input schedule in linear time. We will discuss the generation of the input schedule for horizontal and vertical data separately. We will first answer the question: given a set of horizontal paths, how are we going to generate the optimal set of vertical paths by deciding on where they should be sent in (vertical data input schedule)?

Since the routing of the data paths inside the systolic array is managed locally at individual cells, we can just send in some initialization data and wait for the systolic array to indicate the best input ports to be used. During such an initialization phase, vertical initialization data are sent in at all external vertical input ports. By making use of the sr_o signals generated, we can determine the optimal vertical input schedule. Fig. 11 depicts the process of vertical path initialization. During initialization, the sr_o signal generated from a cell on the top row is used to "*kill*" the data token that enters the array at the next column. If the number of vertical paths that successfully pass through the array is greater than or equal to the desired number of paths, say n, then the reconfiguration is successfully done. The leftmost n input ports that do not have their input data "killed" by the cell on the left form the desired input ports. It is obvious that there is no unused cell on the top row in between the first and the n-th vertical paths. Hence, by theorem 2.5, the set of non-intersecting vertical paths obtained is optimal.

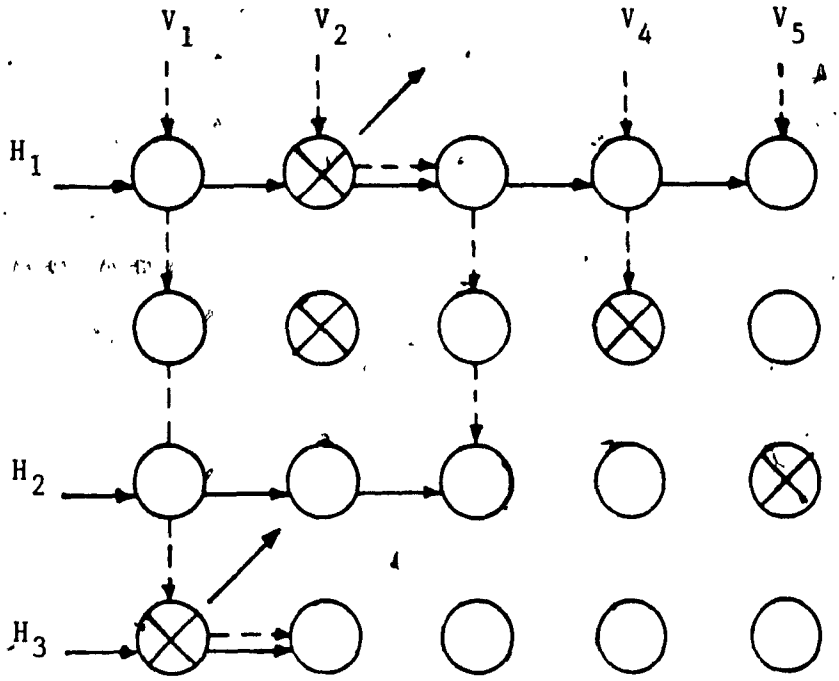


The second row is removed.

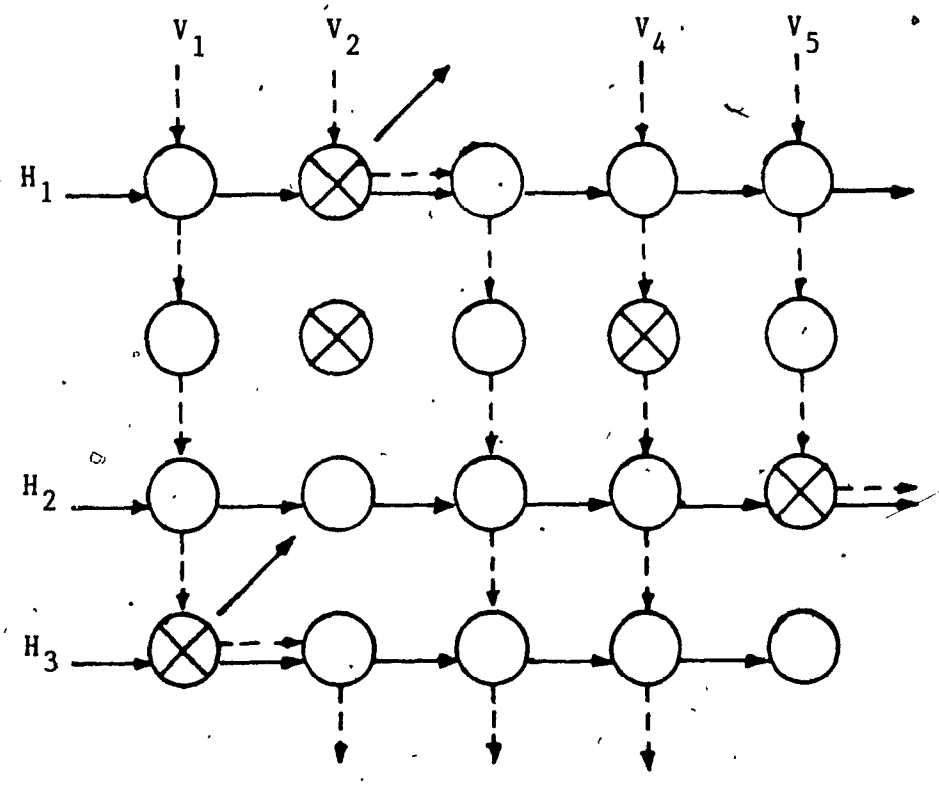
(a) T = 3; three cycles after initialization started.



(b) T = 4. Data on V_3 is killed by the SR_o signal generated by the second cell on the first row.



(c) T = 5.



(d) paths V_1, V_2 and V_4 can pass through the array

Fig. 11 Vertical paths initialization

The second question we have to answer is: how are we going to select the m out of M horizontal rows? Since no global information is available, a simple heuristic is applied in solving the problem. The M-*m* rows to be removed are selected one by one. In every iteration, the row that contains the maximum number of faulty cells will be identified and removed by not sending horizontal data in that row. Direct implementation of simple *count-and-compare* function will be too costly. We solve the problem in a systolic manner. The count-and-compare function is implemented by the *racing* of the sp signals (fig. 12) that propagate through the array from right to left in the rows. Different delays are introduced at a good cell (1 cell/2 cycles) and at a faulty cell (1 cell/cycle). The row that wins the race will contain the maximum number of faulty cells and will be removed. It is also required that the winner identifies itself without an external monitor. To achieve this, a *time skewing* is introduced into the race. The i-th row (counted from the top) starts the propagation of the sp signal at the i-th cycle (if the row is active). When the sp signal of each row reaches the left, the row will claim itself as the winner of the race and notify the other rows (if it has not been disabled to do so) by:

1. sending a signal to *disable* the rows below it from making a claim, and
2. sending a signal to *nullify* the claims of the rows above it.

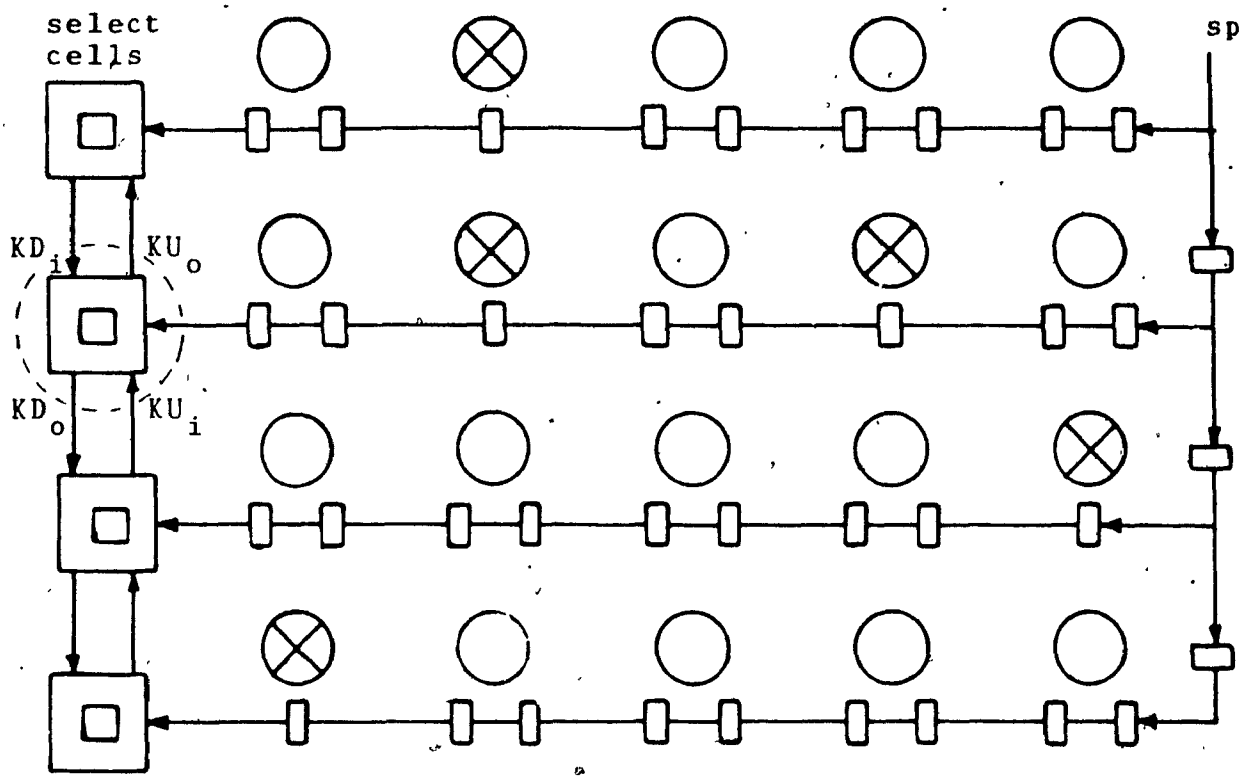


Fig. 12 Row elimination by racing of the sp signal. The second row will win the race and be removed.

The details of the scheme now follow. The sp signal of the cell at the top right corner will be set to 1. This signal will propagate downward (1 row/cycle) and to the left in each active row. The winner of the race is identified by a column of "select" cells (fig. 12) on the left of the processing array. Each select cell has a latch to store the sp signal and four "kill" signals, KD_i, KD_o, KU_i, and KU_o. All

the four kill signals and the latch of each select cell are reset before each iteration takes place. The operation of the cell is as follows:

case 1. $KD_i = 0$, $KU_i = 0$, and $sp = 1$:

The sp signal will be latched. The KU_o and KD_o signals will be set to 1 and are propagated at a speed of 1 cell/cycle upward and downward, respectively. The top and bottom cells, however, will only generate the KD_o and KU_o, respectively.

case 2. $KD_i = 1$ and $KU_i = 0$:

The latch will remain in its current state. The incoming sp signal will be ignored. The KD_o signal will be set to 1.

case 3. $KU_i = 1$:

The latch will be reset to 0. KU_o will be set to 1. If $KD_i = 1$, then KD_o will be set to 1; else KD_o will stay in its current state.

The winner of the race can be identified within 2(N+M) cycles. The bottom row will start the propagation of the sp signal at the M-th cycle. The sp signal takes at most 2N cycles to reath the left hand side, and at most another M cycles are required to identify the most faulty row. After 2(N+M) cycles, the row with the latch value of 1 contains the maximum number of faulty cells.

In practice, the initialization of the horizontal and vertical paths can be done together. At the beginning, test

data are sent in at all input ports. The number of vertical data that exit the array from the bottom after N+M cycles is equal to the number of vertical paths that successfully pass through the array. If there are at least n vertical paths that pass through the array, then the initialization is done. If less than n vertical paths can pass through, then one horizontal row will be removed if the number of horizontal paths is greater than m; otherwise the array cannot be reconfigured into a m x n array using the distributed approach. The most faulty row can be identified in 2(N+M) cycles. The row identified will be removed from future computation. The total time, T, required to initialize the array is:

$$T < (N+M) + (M-m)((2N+2M) + (N+M))$$
$$= (N+M) + 3(M-m)(N+M)$$
$$= (3(M-m)+1)(N+M)$$

For $M-m \ll N$ or M, the initialization can be done in approximately linear time. The mechanism elaborated so far still suffers from having to choose I/O ports for the array. Chapter 3 will remedy this drawback with an appropriate design of a universal distribution and collection network that hides all such detail from the host (user).

Chapter 3

DATA COLLECTION AND DISTRIBUTION

Because of the variability of the data paths in case new faults arise after initialization, the data that belong to the same path may come out from different output ports. Also, the timing relationship between data of different paths is complicated. This imposes difficulty in collecting the output data. Moreover, the vertical path initialization process described in section 2.3.2 requires the user to select proper input ports to send in data. To make the configuration totally transparent to the user, a universal interface is introduced, which will allow the user to send in and collect data from fixed I/O ports at fixed times independent of the fault distribution.

3.1 Data collection network

The problem of data collection arises because the path lengths of the data are not the same and they can be changed dynamically. If we can somehow equalize these path lengths independent of the fault distribution, then we can expect the output data to come out from fixed output ports at fixed times. A systolic routing network (collection network) is introduced to perform this job. Two types of routing cells are required and their routing functions are shown in fig.

13. It is obvious that the two types of routing cells will preserve the first invariant; that is, two adjacent horizontal (vertical) paths do not cross.

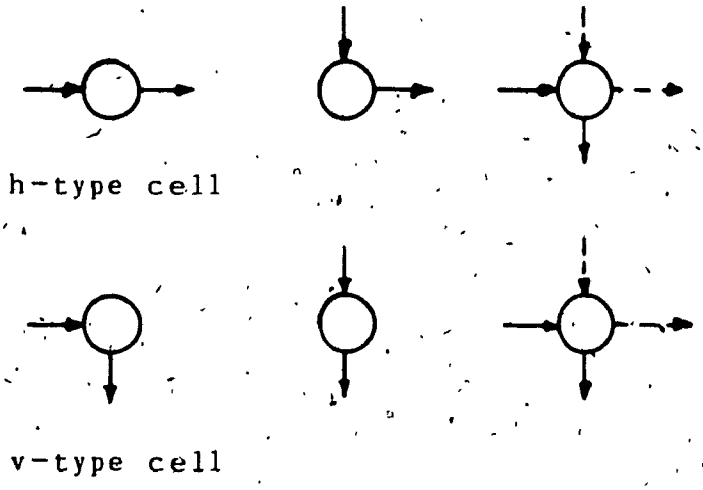


Fig. 13 Routing cells

Fig. 14 shows the structure of the vertical data collection network. The routing network consists of a nxN rectangular array of h-type cells and a nxn square array of v-type cells. The number of rows required in the routing network is equal to the number of vertical paths. The data are greedily routed to the right by the h-type cells and then downward by the v-type cells. By enforcing the first invariant, the i-th vertical path always comes out from the i-th column of v-type cells. The total path lengths (including the time delay introduced at the input of the processing array to satisfy the data skewing requirement) of

all vertical paths are the same. Similarly, a collection network for horizontal data collection can be constructed using a Mxm rectangular array of v-type cells and a mxm square array of h-type cells.

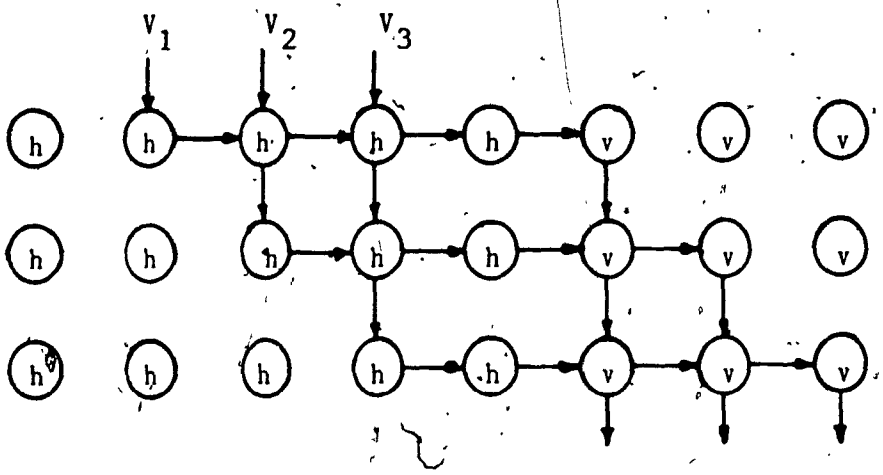


Fig. 14 Vertical data collection network for the example shown in fig. 10.

3.2 Data distribution network

Extending the same idea, we can build similar routing network to distribute the data to the desired input ports of the processing array from fixed external input ports for the DR2 design. In this application, the routing cells should be self-programmable. Fig. 15 depicts the structure of the distribution network for horizontal data. A row of routing cells in the routing network are automatically programmed as

v-type if the corresponding row in the processing array is "removed" by the sp race; else they are set to function as h-type. The programming signals are derived from the "select" cells during initialization. M-m columns of routing cells are required. Similarly, the distribution network for vertical data involves N-n rows. The sr_o signal from the top row are used to program the routing cells when initialization is done. The j-th column of routing cells are set to function as h-type if the sro signal of the processing cell on the top row of column j-1 is set, else they are set to function as v-type. With the help of the self-programmable distribution networks, the configuration of the processing array is totally transparent to the uset.

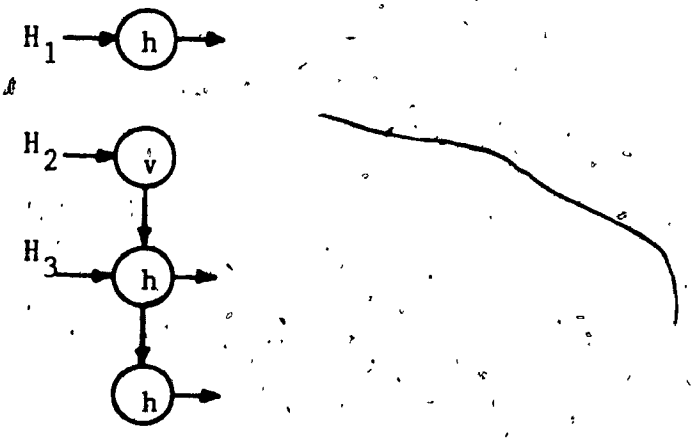


Fig. 15 Horizontal data distribution network for the example shown in fig. 10.

Chapter 4

COVERAGE OF TRANSIENT FAULTS

In the two designs described in chapter 2, the routing of the data paths is based on local information only. No fixed paths are assigned to the horizontal and/or vertical data. If new faults (transient or permanent) arise during computation, the data can be re-routed dynamically to an adjacent cell to retry the computation. So long as there are sufficient spare cells to replace the faulty cells, correct computation can be obtained as shown now.

4.1 Coverage of Single Transient Fault

In this section, we assume that only a single transient fault occurs during the computation. The amount of redundancy required to ensure the coverage of one transient fault for the two designs described in chapter 2 will be established below.

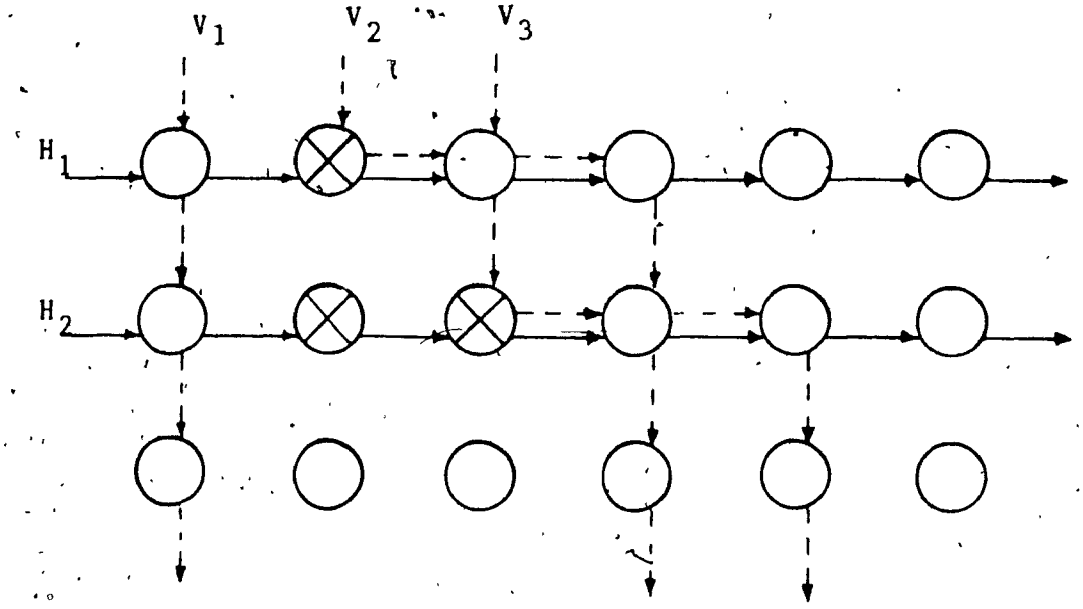
Theorem 4.1

Even if the input schedule is not changed, an array of type-B cells can tolerate one additional fault (transient or permanent) independent of the fault location after initialization, provided there is a spare (unused) vertical path on the right of V_n.

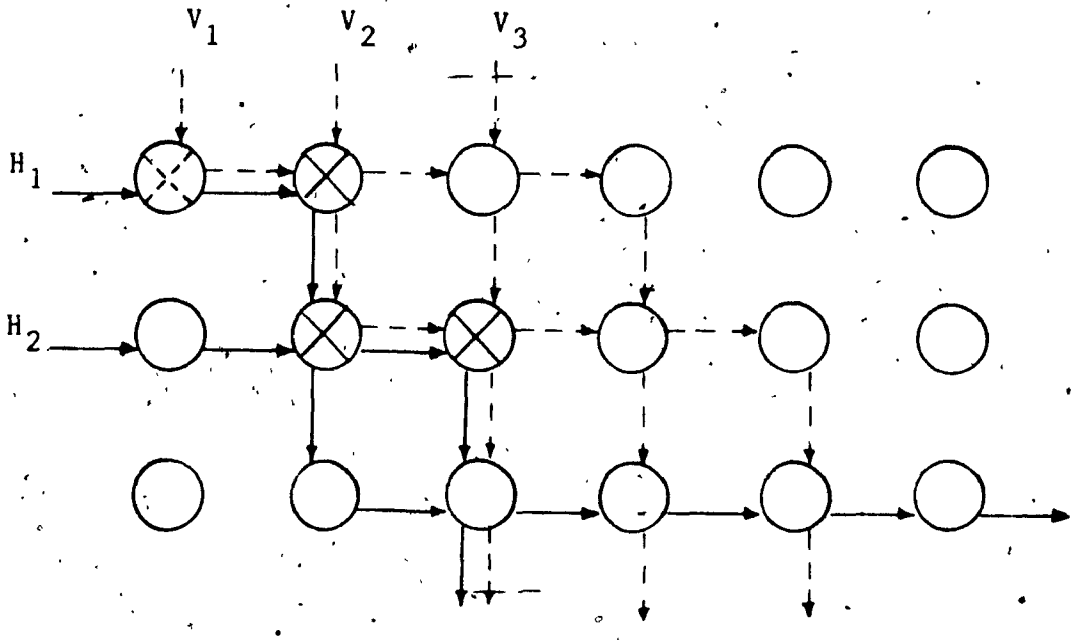
Proof

If one additional fault occurs at an active computation site during the computation, the vertical data will be re-routed locally and dynamically. Since all the vertical paths are initially non-touching, the addition of one fault may only lead to the touching of two adjacent vertical paths. Hence the failure cases indicated in routing table 4 which involve touching of three vertical paths can never occur. If the new fault causes V_j to touch V_{j+1}, V_{j+1} will then be displaced to the right. The new route of V_j will lie on or to the left of the old route of V_{j+1}. Consequently, V_{j+1} may touch V_{j+2} and causes V_{j+2} to be displaced to the right; and so on. This displacement will not cause V_n to miss Hm as long as there is an additional spare path on the right of V_n. **Q.E.D.**

For the DR1 design, both the horizontal and vertical paths are reconfigurable. Intuitively, one spare row and one spare column of non-faulty cells seem to be sufficient to ensure the coverage of one additional fault. Fig. 16 shows a counter example which disproves this statement. In the worst case, m spare rows and one spare column are required to ensure the coverage of one additional fault. Consider the fault distribution shown in fig. 17. Suppose the inputs are packed towards the top-left corner. If the cell at the top-left corner becomes faulty, then the first horizontal path is going to take m steps downward. Conse-



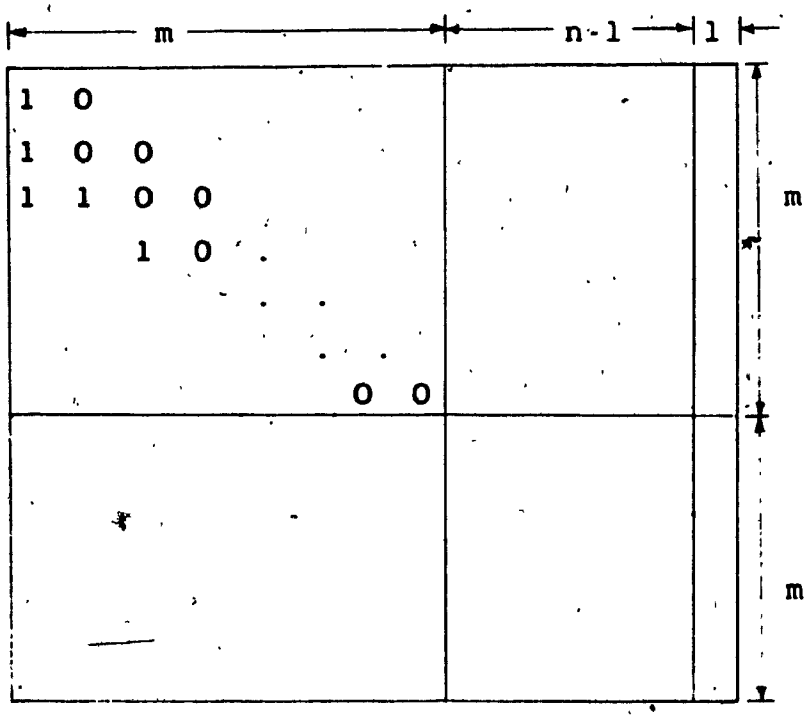
a) a restructured 2x3 array of type-B cells with one spare row and one spare column



b) new fault occurs in the cell at the top-left corner

Fig. 16 Counter example showing that one spare row and one spare column are not sufficient to ensure the coverage of one new fault.

quently, m spare rows are required. The superiority of DR2 to DR1 is even more pronounced if multiple new faults are to be tolerated.



0 : faulty cell
1 : non-faulty cell
All cells not labelled are non-faulty

Fig. 17 Worst case fault distribution that requires m spare rows and 1 spare column to tolerate a transient fault that occurs at the cell at the top-left corner.

4.2 Coverage of Multiple Transient Faults

When dealing with multiple transient faults, the input schedule for the DR2 design has to be adjusted in accordance with the occurence of new faults to maintain the non-touching property of the vertical paths. This is accomplished by the use of the data distribution network described in section 3.2. Since the distribution network is self-programmable, it is possible to modify the vertical data input schedule (for the DR2 design) without interrupting the computation. If the set of vertical paths can be made non-touching (by changing the input schedule) even if a new fault arises during computation, then multiple additional faults can be tolerated.

Theorem 4.2

If the routing cells in the vertical data distribution network (for the DR2 design) are programmable during computation (by the sr_o signal), then the processing array is able to tolerate one additional fault per spare vertical path on the right provided these new faults arise at least N+M cycles apart.

Proof

It has been shown in the proof of theorem 4.1 that if the vertical paths are non-touching, the processing array can always tolerate one additional fault provided there is a spare vertical path on the right. If a new fault is

detected, a chain of sr_o signals are generated. When the sr_o signal reaches the top row, the corresponding column of routing cells are set to function as h-type one by one. The internal input schedule of the processing array is adjusted to take care of the new fault. The rippling of the data dies out within N+M cycles (M cycles for the sr_o signal to reach the top row, N-M cycles for the rippling of the data on the top row, and another M cycles for the last vertical path to take up a new stable route), and a new set of non-touching vertical paths is obtained. Hence, the proof.

Q.E.D.

If the processing cell in the systolic array contains local storages (state of the processor), then the temporary fault should be treated as if it were a new permanent fault (remain in a faulty state) until the array is re-initialized. The reason for this is that, when the data are re-routed, the local storages should also be routed to the new computation site. When the temporary fault disappears later, the original configuration cannot be restored instantaneously as it requires bringing back the local storages to the original cell and re-alignment of future data, features not supported by the architecture presented.

Chapter 5

PERFORMANCE EVALUATIONS

The performance of the distributed reconfiguration approach is studied from both theoretical and practical points of view. First we try to answer the following two questions:

1. *Fault tolerance*: how many faults, regardless of their locations in the faulty array, are definitely tolerable?
2. *Cell redundancy for successful reconfiguration*: asymptotically how much cell redundancy is needed in the faulty array to ensure successful reconfiguration?

The above two fault-tolerance characteristics of two static reconfiguration approaches, namely, the classical row-column elimination approach [7] (referred to as RC-cut), and Kung and Lam's approach [19] (referred to as RCS-cut), have been studied in depth by Lam et al [24]. The results derived in [24] will be extended to our distributed reconfiguration approach. Secondly, practical comparison of the distributed reconfiguration approach with the RC-cut and the RCS-cut approaches that derive the data paths with the global knowledge of fault distribution is studied via computer simulation.

5.1 Fault Tolerance

In this section, we want to extend the lower bound results on the number of faults that are definietly tolerable reported in [24] to our distributed reconfiguration architectures. In particular, we will consider reconfiguring a $2n \times 2n$ array into a $n \times n$ array.

Definitions:

1. An array is said to be k-fault tolerant if it can tolerate k arbitrarily distributed faults.
2. An array is said to be exactly k-fault tolerant if it is k-fault tolerant but not (k+1)-fault tolerant.

We will show that both the DR1 and DR2 designs are exactly $3n$-fault tolerant when reconfiguring a $2n \times 2n$ array into a $n \times n$ array.

Lemma 5.1

Any solution of the RC-cut approach is a feasible solution in the DR1 design if the data input schedule is derived by the user based on the fault distribution.

Proof

If the data are input at exactly the same places as in the RC-cut approach, then the configuration of the array obtained by the distributed reconfiguration algorithm will be the same as that of the RC-cut. This is because each horizontal (vertical) path is going to meet every vertical (horizontal) path at an unused good cell. The two paths

will cross each other right after passing through the cell, hence no pairing of paths will occur. Q.E.D.

Theorem 5.1

A $2n \times 2n$ array of type-A cells is exactly $3n$-fault tolerant when reconfigured into a $n \times n$ array using the distributed approach if the input schedule is derived from the fault distribution.

Proof

It has been proved in [24] that a $2n \times 2n$ array is exactly $3n$-fault tolerant when reconfigured into a $n \times n$ array using both the RC-cut and the RCS-cut approaches. According to lemma 1, the search space of the distributed approach (with offline analysis) is larger than that of the RC-cut approach, however, it is smaller than that of the RCS-cut approach. The proof now follows since both the RC-cut and RCS-cut approaches are exactly $3n$-fault tolerant.

Q.E.D.

Lemma 5.2

A $2n \times 2n$ array of type-B cells is $3n$-fault tolerant when reconfigured into a $n \times n$ array using the distributed approach even if the input schedule is self-generated using the method described in section 2.3.2.

Proof

Suppose there are $3n$ faults in the array. The sp row elimination mechanism can remove the n most faulty rows cov-

ering at least $2n$ faults, leaving behind at least n fault-free columns which can be detected by the vertical path initialization process. Hence, $3n$ faults can always be tolerated. Q.E.D.

Theorem 5.2

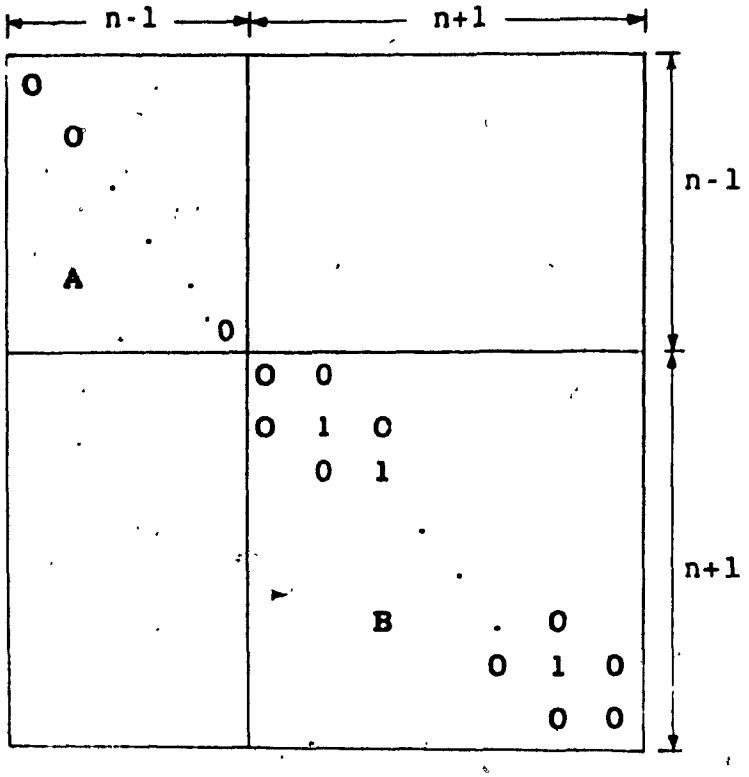
A $2n \times 2n$ array of type-B cells is exactly $3n$-fault tolerant when reconfigured into a $n \times n$ array using the distributed approach with self-generation of input schedule.

Proof

According to lemma 5.2, the array can tolerate at least $3n$ faults. An example of an irrecoverable $2n \times 2n$ array having $3n+1$ faults (fig. 18) has been reported in [24]. In the distributed approach, the n most faulty rows are removed to cover $2n$ faults in region B. There remains $n+1$ faults distributed in $n+1$ columns, none of which can share a vertical path with another. Hence, only $n-1$ vertical paths can be constructed. Q.E.D.

5.2 Asymptotic Redundancy Requirement

Here we consider the probabilistic reconfigurability of the distributed reconfiguration approach. In particular, we try to establish the asymptotic amount of redundancy represented by the number of additional cells needed to ensure successful reconfiguration of a given $M \times N$ array into a $n \times n$ array. Let $F[M \times N \rightarrow n \times n]$ denote the proba-



0 : faulty cell
1 : non-faulty cell
All cells not labelled are non-faulty

Fig. 18 A 2n x 2n irrecoverable array with 3n+1 faults

bility of failure to reconfigure a M x N array into a $n \times n$ array. It has been shown in [24] that for the simplified RCS-cut approach in which the horizontal paths are fixed as the physical rows and the vertical paths are constructed such that after the j-th vertical path has been constructed using the first i columns, the (j+1)-th vertical path will be constructed greedily starting with the (i+1)-th

column,

$$\lim_{n \to \infty} F[n \times O(n^2) \to n \times n] \to 0$$

For the DR1 design, if the input schedule is derived from the global knowledge of the fault distribution, then any solution of the simplified RCS-cut approach is also a feasible solution of the DR1 design. Hence, $O(n^3)$ redundancy is asymptotically sufficient for the DR1 design.

For the DR2 design, the search space of the vertical path generation mechanism is larger than that of the simplified RCS-cut approach. Hence, $O(n^3)$ redundancy is also asymptotically sufficient for the DR2 design.

5.3 Computer Simulation

The reconfigurability of a faulty array when using different reconfiguration algorithms is studied via computer simulation. The utilization of the array (ratio of mn/MN) is used as the figure of merit. The performances of the distributed algorithms described in chapter 2 are compared with two other reconfiguration algorithms. The four reconfiguration algorithms are:

1. DR1 -- The distributed algorithm described in section 2.2. The input data are assumed to be packed towards the top-left corner.
2. DR2 -- The distributed algorithm described in section

2.3.

3. RC -- The classical row-column elimination approach [7]. Solving for the global optimum solution of this problem is equivalent to the maximum node matching problem of a bipartite graph [21], which is NP-complete. In the simulation study, we solve for a local optimum using a simple heuristic by alternately removing the most faulty row and the most faulty column. See appendix 1 for details of the algorithm.

4. RCS -- This follows Kung and Lam's approach [19]. The program implements the algorithm proposed by Li et al [25]. This algorithm will maximize the number of vertical paths obtainable for a given number of horizontal paths required. Assume the required working array size is $m \times n$. The horizontal paths H_1, H_2, ..., H_m are generated successively. Assume H_1, H_2, ..., H_{i-1} have been generated and H_i is to be generated. H_i will be constructed starting from the first (highest) cell in column 1 of the original array below H_{i-1}. H_i is then greedily moved horizontally to the right along that row. A corresponding optimal set of vertical paths V_1, V_2, ..., V_p will then be generated. If $p > n$ then the program will proceed to construct H_{i+1}; else a different H_i is generated by backtracking along H_i until encountering a faulty cell at which a turn was not taken previously, and then extending the new Hi by taking a turn at the faulty cell. If the backtracking ends at the first cell

in H_i, then the existing H_i must be discarded and the generation of H_i is restarted from the next row. Backtracking to H_{i-1} will be required if all possible combination of H_i have been enumerated. The complexity of this algorithm is $O(2^k MN)$ where k is the number of faulty cells in the array. The performance of this algorithm is the theoretical upper bound of the two distributed algorithms.

For all the four algorithms, a local/global optimum is obtained iteratively depending on the defined objective. In each iteration, a certain number of horizontal paths, say m, is assumed, and the number of vertical paths obtainable is maximized. For the DR1, the optimum solution is obtained by extensive enumeration. The value of m is changed from 0.7M (this value is determined by some test runs of the program) to M and the highest score (value of mn) is recorded. For the other three algorithms, the program starts with an initial guess (close to optimum), say $m1$, based on some test runs of the program. If the score of the case $m = m1+1$ is higher than that of $m = m1$, then the search will continue by increasing m; else the search will continue by decreasing m. The program stops when the score of the current trial is less than that of the previous trial.

Faults are randomly injected into the array with two distribution functions:

(1) Uniform

Two uniformly distributed random numbers ranging from 0 to 1 are generated. The x- and y-coordinates of the fault location is obtained by scaling the two random numbers by N and M, respectively. If the cell location obtained has already been marked faulty, then another location will be generated using the same procedure.

(2) Clustered

Each cluster of faults is generated as follows : the cluster center (x_0, y_0) is generated as a uniformly distributed fault. The other faults are generated with respect to the cluster center using the polar method. Let r be the distance from the cluster center and θ the angle with the x-axis. Then the fault location is determined by

$$x = x_0 + r\cos\theta,$$

$$y = y_0 + r\sin\theta$$

where θ is a uniformly distributed random number ranging from 0 to 2π and r is a Gaussian distributed random number with the mean value proportional to the square root of the cluster size. The routine to generate r can be found in appendix 2. If the determined location is already marked faulty or falls outside the physical array then it will be replaced by another one generated in the same way. To avoid getting a very large cluster, the cluster size is limited to 40. If the number of faults to be generated is greater than 40, then two

equally sized clusters (with distinct cluster centers) are generated.

Example fault distribution patterns can be found in appendix 3. For each case, fifty different fault distribution patterns are simulated and the mean values of the performances are shown in the plots in figs. 19 to 21. The variances of the data samples of figure 19 are shown in tables 5 and 6. The simulation results are summarized below:

1. The performances of the two distributed approaches when reconfiguring a 32 x 32 array are close to that of the RCS for both uniformly distributed and clustered faults (fig. 19).
2. As expected, RC has better performance when dealing with clustered faults (fig. 19), however, it is still much inferior to the other three algorithms. Note that the difference in performance of the RC as compared to the others increases with the array size.
3. One interesting finding is that all the algorithms, except DR1, have better performances when restructuring rectangular arrays than the square counterpart (fig. 20). The exception of DR1 is mainly due to the sensitivity of the algorithm to the faults in the top left quadrant.
4. Fig. 21 shows the asymptotic behavior of DR1, DR2 and RC. For a given fixed percentage of faults, the two distributed algorithms are found to have a steady utili-

zation as the size of the array increases. No similar trend is observed for the RC. The simulation result seems to imply that $O(n^2)$ redundancy is sufficient to ensure successful reconfiguration, although such a theoretical bound has not been obtained.

When comparing the two distributed algorithms, DR1 has a more flexible routing strategy than DR2 (fixed horizontal paths). However, simulation reveals that confining the horizontal paths does not affect the performance significantly so long as the vertical paths are reconfigurable. Moreover, DR2 has the following advantages over DR1:

1. Self-generation of a good data input schedule.
2. Little sensitivity to the fault distribution.
3. Less redundancy requirement to ensure the coverage of transient faults.

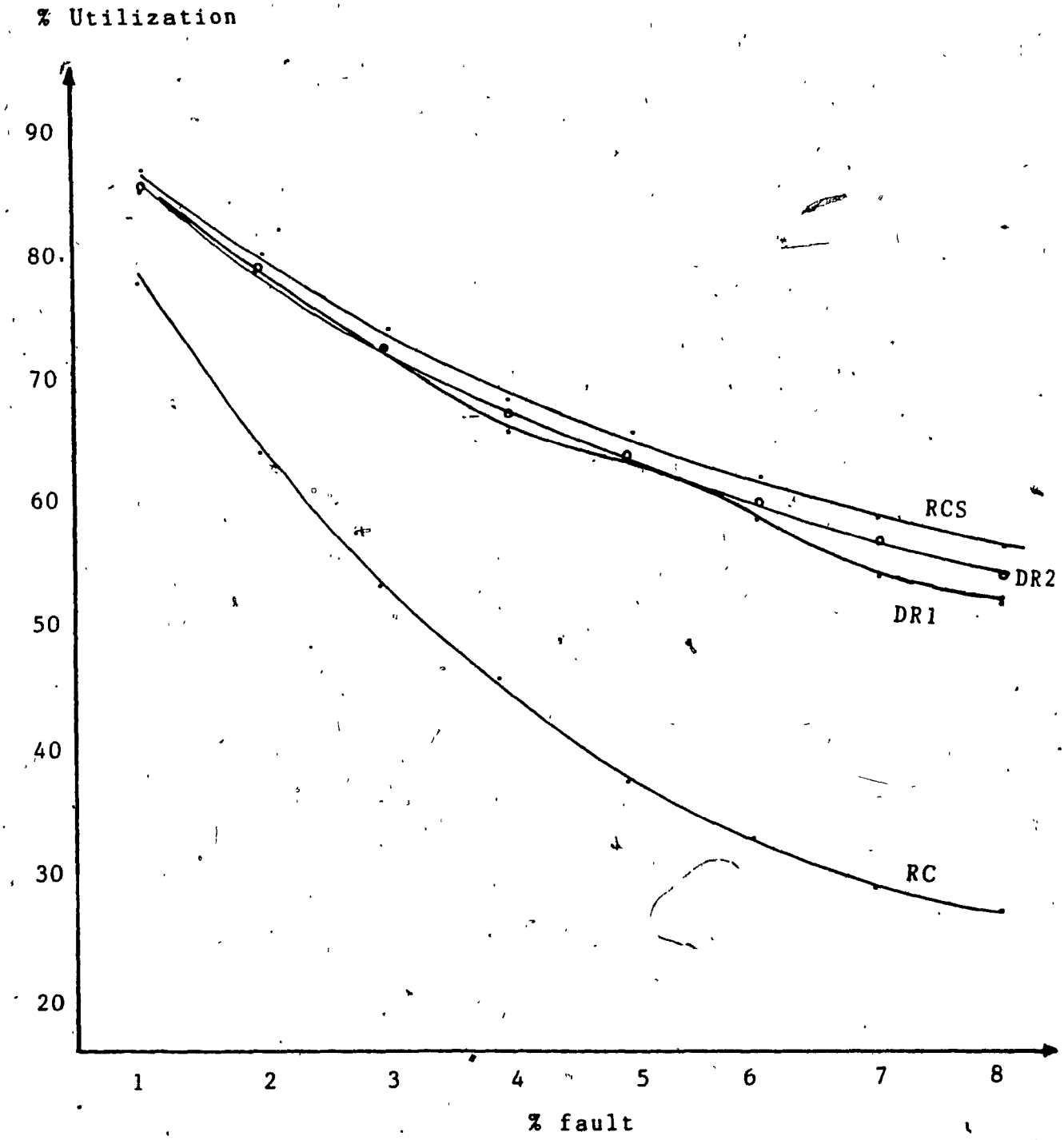


Fig. 19a Performances of the four algorithms when restructuring a 32x32 array with uniformly distributed faults.

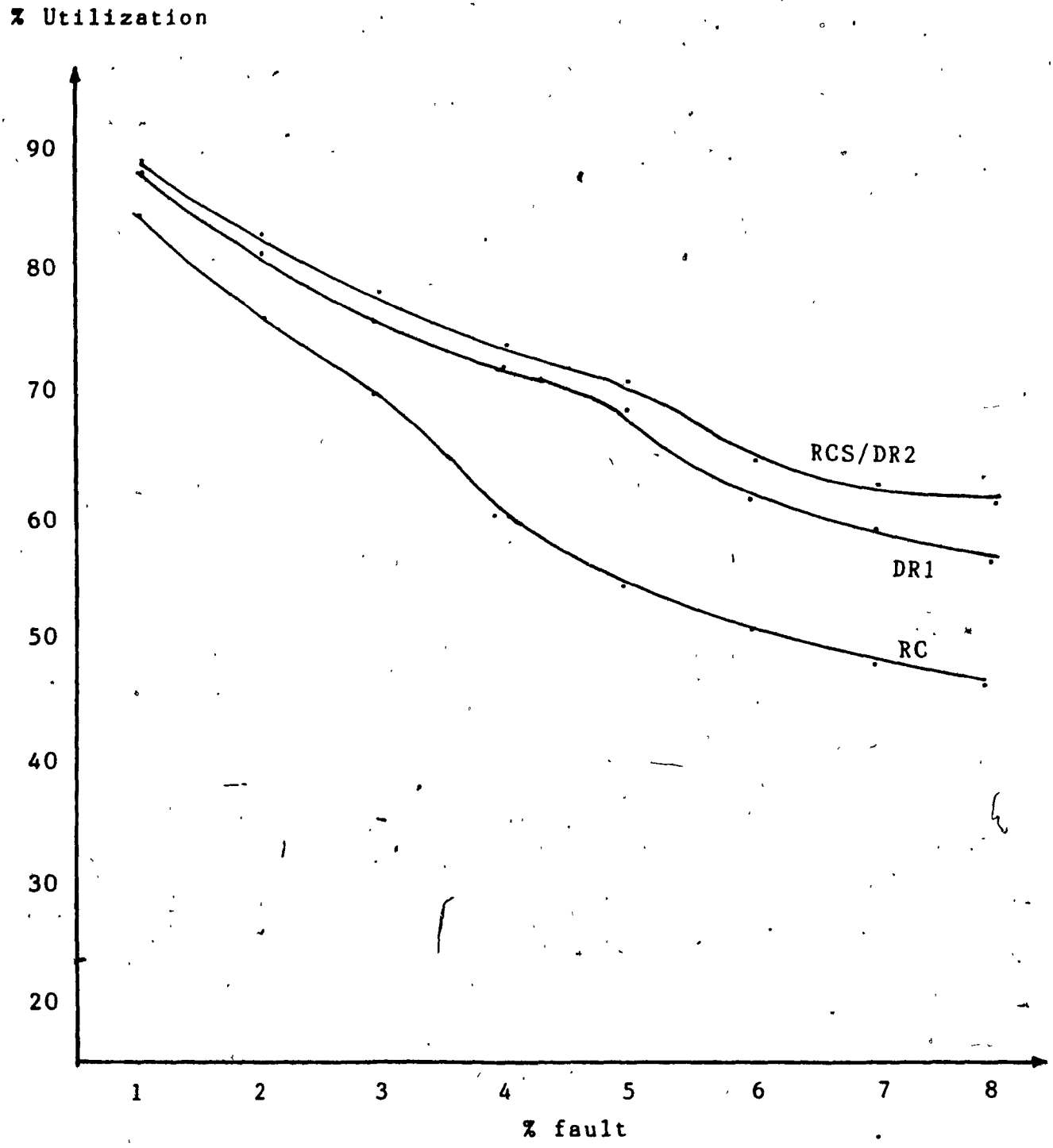


Fig. 19b Performances of the four algorithms when restructuring a 32x32 array with clustered faults.

%fault / algorithm	1	2	3	4	5	6	7	8
DR1	10.5	11.4	19.3	27.3	24.5	19.0	25.3	23.5
DR2	6.8	7.0	10.8	10.2	14.5	11.4	16.0	13.3
RC	7.3	10.6	15.3	21.9	29.9	37.4	37.0	23.0
RCS	3.9	8.5	5.2	12.1	11.4	8.6	12.1	14.5

Table 5 Variances of the samples obtained in the simulation of the performances of the four algorithms with uniformly distributed faults.

%fault / algorithm	1	2	3	4	5	6	7	8
DR1	6.1	9.5	11.2	43.1	45.4	50.1	60.3	50.7
DR2	4.6	5.5	5.5	21.6	31.1	35.9	34.5	36.6
RC	5.4	13.7	16.6	33.2	32.2	37.8	49.7	46.5
RCS	3.17	3.98	6.9	26.4	25.1	36.4	36.6	41.5

Table 6 Variances of the samples obtained in the simulation of the performances of the four algorithms with clustered faults.

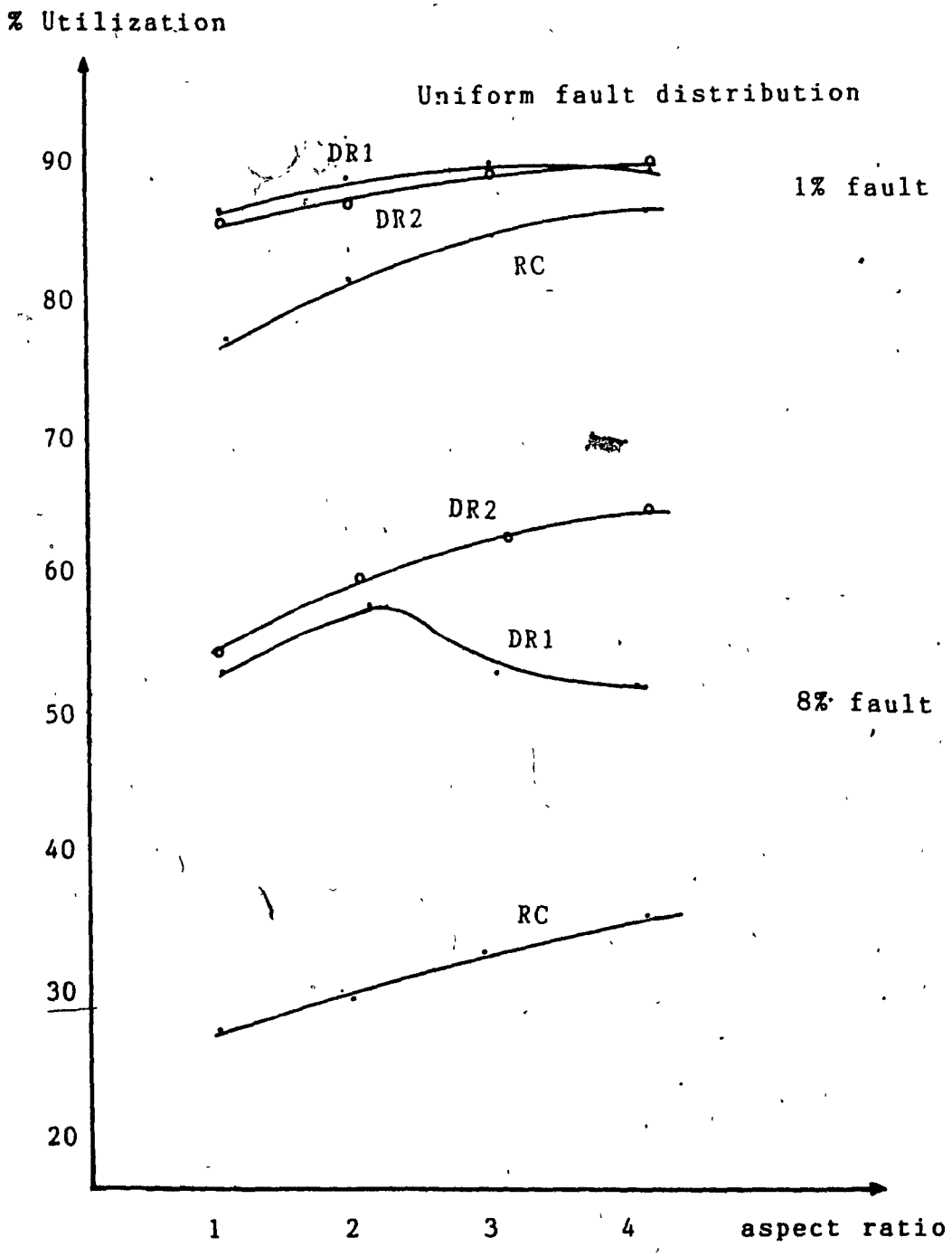


Fig. 20 Plot of performances vs the aspect ratio of the array with fixed array size MN = 1024.

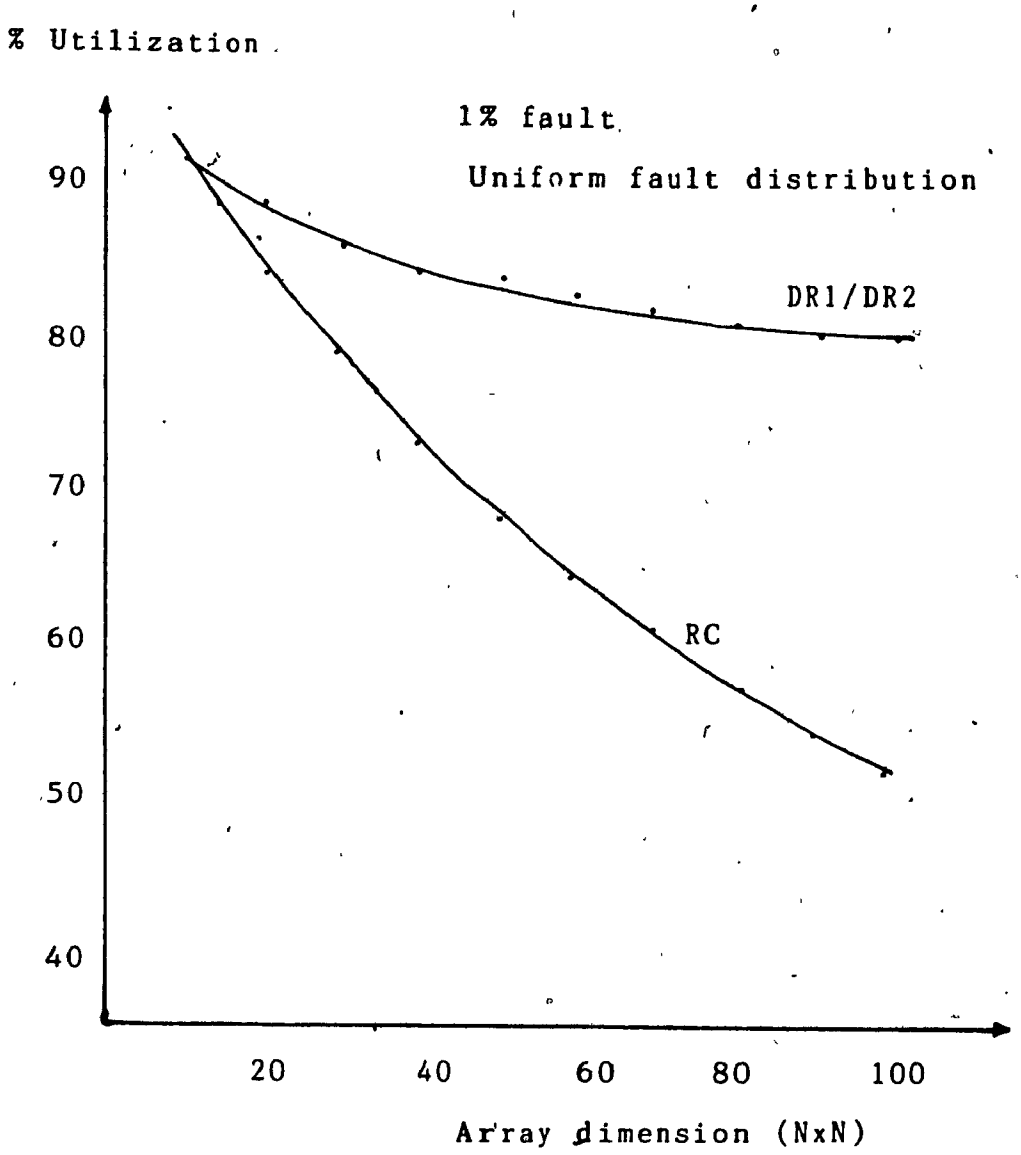


Fig. 21 Asymptotic behaviour of DR1, DR2 and RC with fixed percentage fault.

Chapter 6

EXTENSION TO UNIVERSAL SYSTOLIC ARRAYS

The concept of the distributed reconfiguration approach described in chapter 2 can be extended to universal systolic arrays. 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. Consider the type-C cell shown in fig. 22. The diagonal output is confined to flow diagonally whereas the horizontal (vertical) output can pass through a physical horizontal (vertical) or diagonal edges, that is, the diagonal paths are fixed but the horizontal (vertical) paths are reconfigurable. In each cell, there can be at most one valid horizontal (vertical) input at any one time.

Fig. 23 depicts a restructured array of type-C cells. The horizontal paths are numbered top to bottom, the vertical paths are numbered left to right, and the diagonal paths are numbered bottommost diagonal to topmost diagonal. The following facts can be observed.

1. The horizontal path H_i should meet the vertical path V_j and the diagonal path D_{m+j-i} at a common good cell.
2. The first active cell (where computation is performed) in H_i should be on D_{m-i+1} and the first active cell in V_j should be on D_{m+j-1}.

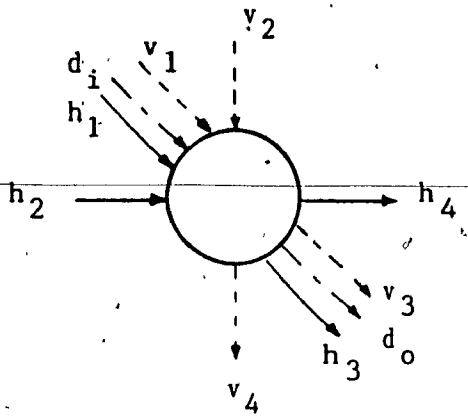


Fig. 22 Type-C cell architecture

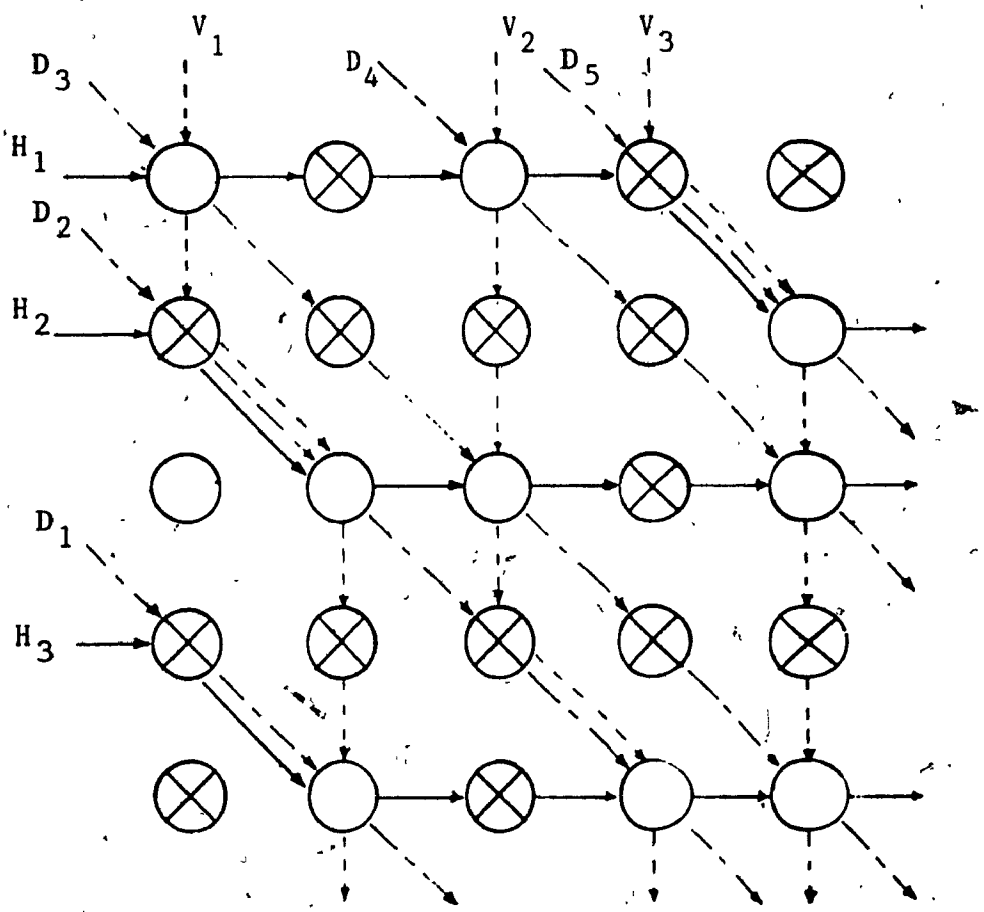


Fig. 23 A restructured array of type-C cells

3. The diagonal path D_i should contain at least min{i, $m+n-i$} good cells in it.

Consider reconfiguring a M x N array of type-C cells into a m x n array. The set of diagonal paths are first identified based on the requirement that D_i should contain at least min{i, $m+n-i$} good cells. The external input port for the horizontal path H_i (vertical path V_j) will be the same as that of D_{m-i+1} (D_{m+j-1}). The routing strategy of the horizontal and vertical paths is as follow.

1. If there is only one path entering a cell, it will be routed to the right, downward or diagonally depending upon whether it is a horizontal, vertical or diagonal path, respectively.
2. H_i meets D_k only. H_i will be routed diagonally along D_k.
3. V_j meets D_k only. V_j will be routed diagonally along D_k.
4. When H_i meets V_j and D_k at a common good cell, the corresponding computation will be performed. The three paths H_i, V_j and D_k will be routed to the right, downward and diagonally, respectively.
5. When H_i meets V_j and D_k at a common faulty cell, the three paths will be routed all together along D_k to find a good cell to perform the required computation.

The above routing function only requires local informations such as the status of the cell (good or faulty) and

the number of data paths entering the cell. This routing function can be implemented in the same way as for the 2-data flow systolic arrays. Detailed studies regarding the automatic selection of a good set of diagonal paths and the performace of the distributed reconfiguration approach when applied to universal systolic array are left to future research.

Chapter 7

SUMMARY AND CONCLUDING REMARKS

A distributed reconfiguration approach based on local invariants technique is presented. The routing of data paths is done locally, and intelligence is incorporated to preserve the invariants. Two cell architectures are proposed and studied. In the first design (DR1), both the horizontal and vertical paths are dynamically reconfigurable. In the second design (DR2), the routing of the horizontal paths is fixed but the vertical paths are reconfigurable. Routing networks made up of two types of simple routing cells can be used to interface the processing array to the external world, so that data can be sent in and collected at fixed I/O ports at fixed times independent of the internal configuration of the array. The theoretical measures of performances of the two designs are listed in table 7.

Comparison of the distributed approach to the RC and RCS is studied via computer simulation. The simulation results are summarized below:

1. Performances of the two distributed algorithms are comparable to that of the static RCS method where global fault distribution is used to derive the reconfiguration.
2. Simple row-column elimination (RC) method is inferior to

the other three algorithms.

3. Except DR1, the reconfiguration algorithms have better performance when restructuring rectangular arrays.
4. $O(n^2)$ redundancy is apparantly asymptotically sufficient to ensure successful reconfiguration for the two distributed algorithms. This is a much tighter bound than what we can prove theoretically.

The performances of the two designs, DR1 and DR2 are comparable, but DR2 has three desirable properties: (1) self-generation of input schedule; (2) little sensitivity to fault distribution; and (3) less redundancy requirement to tolerate transient fault.

Performance	DR1	DR2
$3n$-fault tolerant	yes, with offline analysis	yes
Asymptotic cell redundancy required	$O(n^3)$, with offline analysis simulation indicates $O(n^2)$	$O(n^3)$ simulation indicates $O(n^2)$
Worst case redundancy to cover one additional fault	m spare rows and one spare column	one spare column

Table 7 Theoretical measures of performances of DR1 and DR2.

7.1 Future Research

The effectiveness of the distributed reconfiguration approach depends on how good is the fault detection scheme. Traditional fault detection approaches such as duplicate-and-compare, coding, and recomputation with shifted operands require large amount of hardware and/or time overhead. To reduce the overhead cost, fault detection should be done at the functional level. Further research in developing high (functional) level fault models that can accurately capture the faults at lower (physical) level is required.

The distributed reconfiguration approach has been demonstrated for uni-directional 2-data flow systolic arrays, however, it can be extended to restructure universal systolic arrays. One major problem to be solved is to develop an automatic mechanism to generate the set of diagonal paths.

The distributed algorithms described in chapter 2 only work for uni-directional 2-data flow systolic arrays. Extending the algorithm for bi-directional data flow systolic arrays will be an interesting problem.

REFERENCES

1. Jacob A. Abraham, W. Kent Fuchs, "Fault and Error Models for VLSI", *Proc. IEEE*, May 1986, pp 639-654.

2. M. J. Ashjaee, S. M. Reddy, "On Totally Self-Checking Checkers for Separable Codes", *IEEE Trans. Computers*, Aug. 1977, pp 737-744.

3. A. Avizienis, "Arithmetic Codes: Cost and Effectiveness Studies for Application in Digital Systems Design", *IEEE Trans. Computers*, Nov. 1971, pp 1322-1331.

4. Prithviraj Banerjee, Jacob A. Abraham, "Fault Characterization of VLSI MOS Circuits", *Intl. Conf. Circuits and Computers*, 1982, pp 564-568.

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

6. Yoon-Hwa Choi, Miroslaw Malek, "A Fault-Tolerant VLSI Sorter", *Intl. Conf. on Computer Design*, 1985, pp 510-513.

7. J. A. B. Fortes, C. S. Raghavendra, "Gracefully Degradable Processor Arrays", *IEEE Trans. Computers*, Nov. 1985, pp 1033-1044.

8. D. Fussell, P. Varman, "Fault-Tolerant Wafer-Scale Architectures for VLSI", *9th Intl. Symp. on Computer*

Architecture, 1982, pp 190-198.

9. D. Fussell, P. Varman, "Designing Systolic Algorithms for Fault-Tolerance", *Intl. Conf. Computer Design*, 1984, pp 616-622.

10. Kuang-Hua Huang, Jacob A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", *IEEE Trans. Computers*, June 1984, pp 518-528.

11. N. K. Jha, J. A. Abraham, "Techniques for Efficient MOS Implementation of Totally Self-Checking Checkers", *15th Intl. Symp. Fault-Tolerant Computing*, 1985, pp 430-436.

12. Jing-Yang Jou, Jacob A. Abraham, " Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures", *Proc. IEEE*, May 1986, pp 732-741.

13. Jung-Hwan Kim, Sudhakar M. Reddy, "A Fault-Tolerant Systolic Array Design Using TMR Method", *Intl. Conf. on Computer Design*, 1985, pp 769-773.

14. D. E. Knuth, The Art of Computer Programming Vol. 2: Seminumerical Algorithms, *Addison Wesley*, 1973.

15. Israel Koren, "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Array", *Proc. 8th Intl. Symp. Computer Architecture*, May 1981, pp 442.

16. Israel Koren, M. A. Breuer, "On Area and Yield Considerations for Fault-Tolerant VLSI Processor Arrays",

IEEE Trans. Computers, Jan. 1984, pp 21-27.

17. Israel Koren, D. K. Pradhan, "Introducing Redundancy into VLSI Designs for Yield and Performance Enhancement", *Proc. 15th Annu. Symp. on Fault-Tolerant Computing*, 1985, pp 330-335.

18. H. T. Kung, "Why Systolic Architectures?", *IEEE Computer*, Jan 1982, pp 37-46.

19. H. T. Kung, Monica S. Lam, "Wafer-Scale Integration and Twolevel Pipelined Implementations of Systolic Arrays", *Journal of Parallel and Distributed Computing*, Aug. 1984, pp 32-63.

20. S. Y. Kung, D. D. Souza, J. T. Johl, "On Fault-Tolerance in Array Processors", *Intl. Conf. on Computer Design*, 1985, pp 764-768.

21. Sy-Yen Kuo, W. Kent Fuchs, "Efficient Spare Allocation in Reconfigurable Arrays", *23rd Design Automation Conf.*, 1986, pp 385-390.

22. Frank Thomson Leighton, Charles E. Leiserson, "Wafer-Scale Integration of Systolic Arrays", *IEEE Trans. Computers*, May 1985, pp 448-461.

23. P. K. Lala, Fault Tolerant and Fault Testable Hardware Design, *Prentice-Hall*, 1985.

24. C. Lam, H. F. Li, R. Jayakumar, "A study of Two

Approaches for Reconfiguring Fault-Tolerant Systolic Arrays", *to appear in IEEE Trans. Computers.*

25. H. F. Li, R. Jayakumar, C. Lam, "Restructuring for Fault-Tolerant Systolic Arrays", *submitted to IEEE Trans. Computers*, also available as *Technical Report: CCSD-VLSI-86-1; Computer Science Dept., Concordia University, Montreal, Canada*, 1986.

26. H. F. Li, D. Pao, R. Jayakumar, "Dynamic Reconfiguration for Fault-Tolerant Systolic Arrays", *Proc. Intl. Conf. on Parallel Processing*, 1987, pp 110-113.

27. H. F. Li, D. Pao, R. Jayakumar, "A Transient-Fault Tolerant Self-Reconfigurable Systolic Array and Performance Evaluations", *submitted to Journal of Parallel and Distributed Computing; also to appear in Can. Conf. VLSI*, Oct. 1987, as "A Transient-Fault Tolerant Self-Reconfigurable Systolic Array and Related Performance Comparisons".

28. Tulin Erdim Mangir, Algirdas Avizienis, "Failure Modes for VLSI and Their Effect on Chip Design", *Intl. Conf. Circuits and Computers*, 1980, pp 685-688.

29. C. Mead, L. Conway, Introduction to VLSI Systems, *Addison-Wesley*, 1980.

30. Janak H. Patel, Leona Y. Fung, "Concurrent Error Detec-

tion in ALU's by Recomputing with Shifted Operands", *IEEE Trans. Computers*, July 1982, pp 589-595.

31. Dhiraj K. Pradhan, "Fault-Tolerant Architectures for Multiprocessors and VLSI systems", *13th Intl. Symp. Fault-Tolerant Computing*, 1983, pp 436-441.

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

33. David A. Rennels, "Fault-Tolerant Computing -- Concepts and Examples", *IEEE Trans. Computers*, Dec. 1984, pp 1116-1129.

34. A. L. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Array of Processors", *IEEE Trans. Computers*, Oct. 1983, pp 902-910.

35. Mariagiovanna Sami, Rento Stefanelli, "Reconfigurable architecture for VLSI processing arrays", *National Computer Conf.*, 1983, pp 565-577.

36. John Wakerly, Error Detecting Codes, Self-Checking Circuits and Applications, *Elsevier North-Holland*, 1978.

Appendix 1

Two queues are used to store the number of rows and number of columns, respectively, that contain faulty cell(s). The two queues are sorted in descending order of the number of faulty cells in the row/column that have not been covered. The procedure **ARCE** will remove the most faulty row and column alternately. Suppose m rows are required.

```
ARCE()
{
    initilaize the row_queue and column_queue;
    while there is fault not yet covered
    {
        if number of rows left > m
            remove the most faulty row;
            update column_queue;
            update number of faults covered;
        if column_queue not empty
            remove the most faulty column;
            update row_queue;
            update number of faults covered;
    }
    return;
}
```

Appendix 2

The Gaussian distributed random number *r* is generated using the polar method due to Box and Muller [20]. Let U1 and U2 are uniformly distributed random number ranging from 0 to 1. The number *r* is generated as follows:

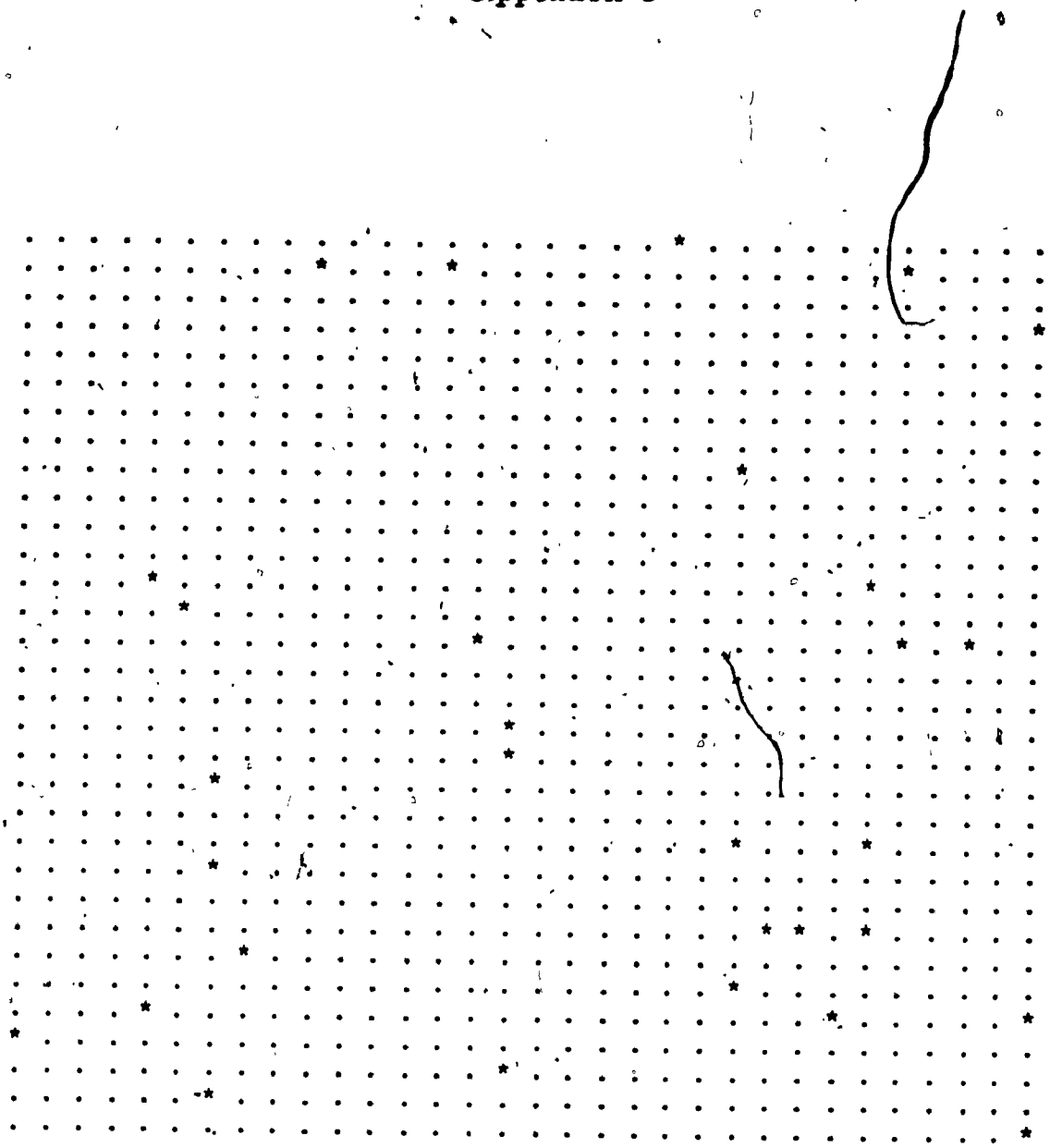
```
double generate_gaussian()
{
    double U1, U2, V1, V2, S, Z;

    do {
        U1 = random();
        U2 = random();
        V1 = 2.0 * U1 - 1.0;
        V2 = 2.0 * U2 - 1.0;
        S = V1 * V1 + V2 * V2;
    } while (S >= 1.0);

    Z = sqrt((-2.0 * log(S)) / S);
    V1 = V1 * Z;
    r = offset + V1 * scale;
    return(r);
}
```

offset is the expected mean value of *r* where

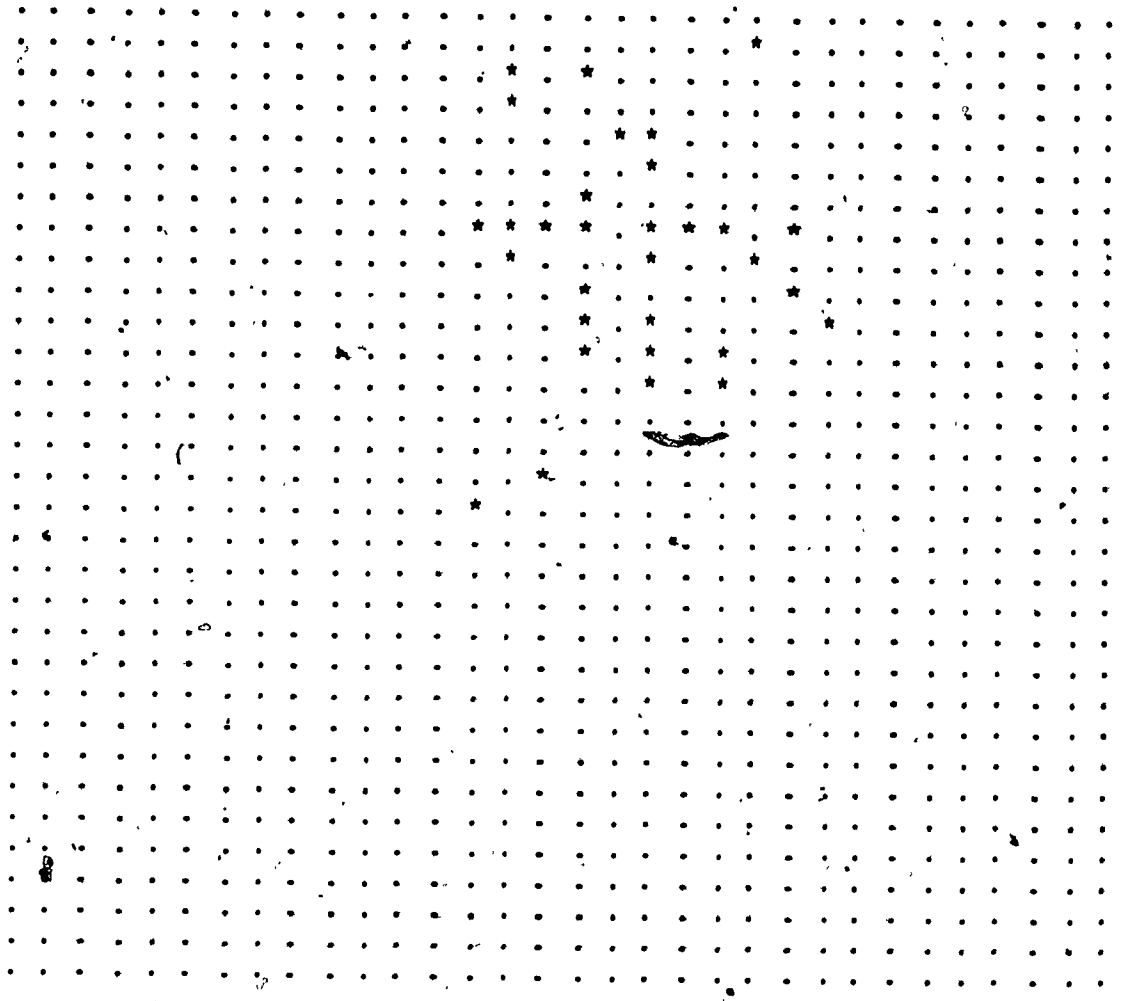
offset = 0.8 * sqrt(cluster_size)

scale is a numerical parameter used to adjust the variance of the samples, and it is set to 3.

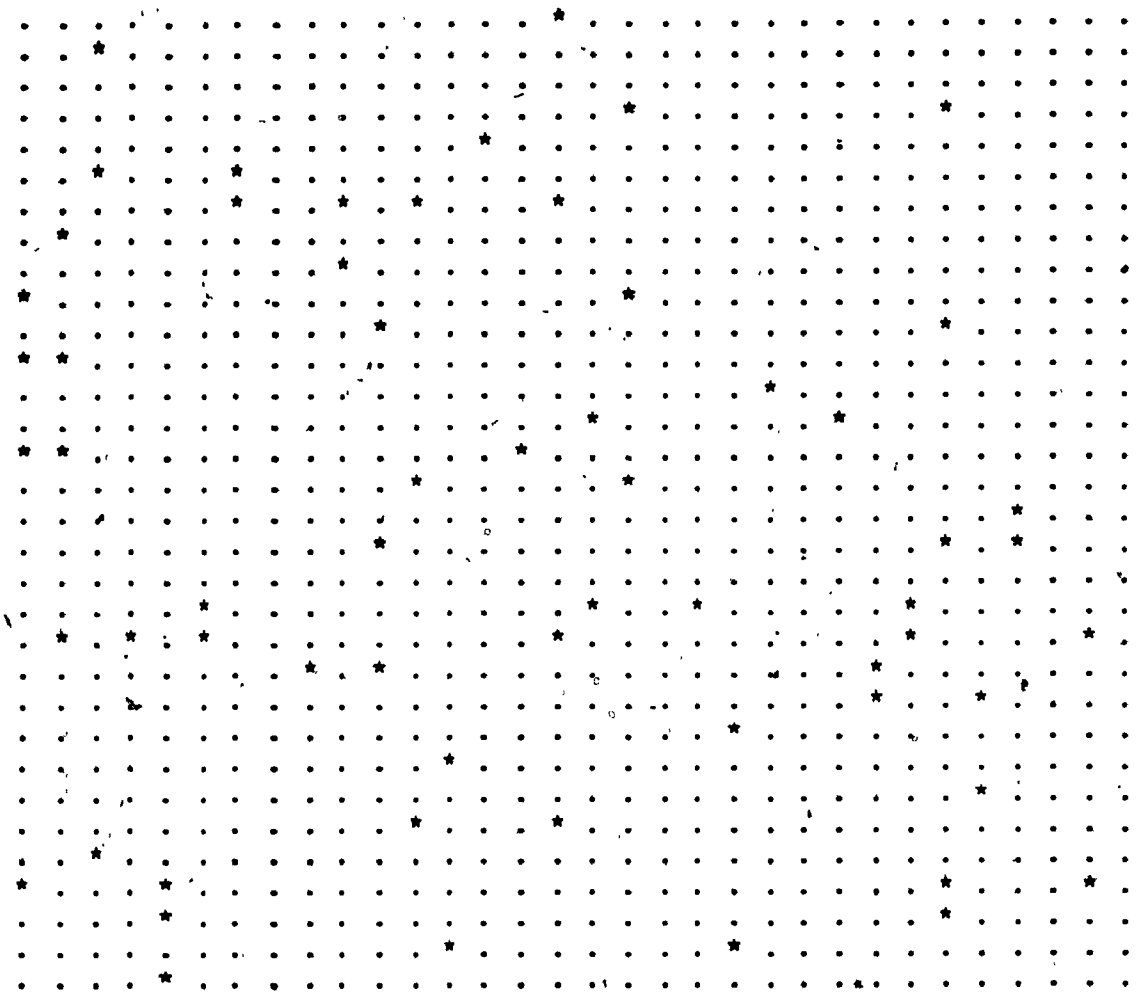
Appendix 3



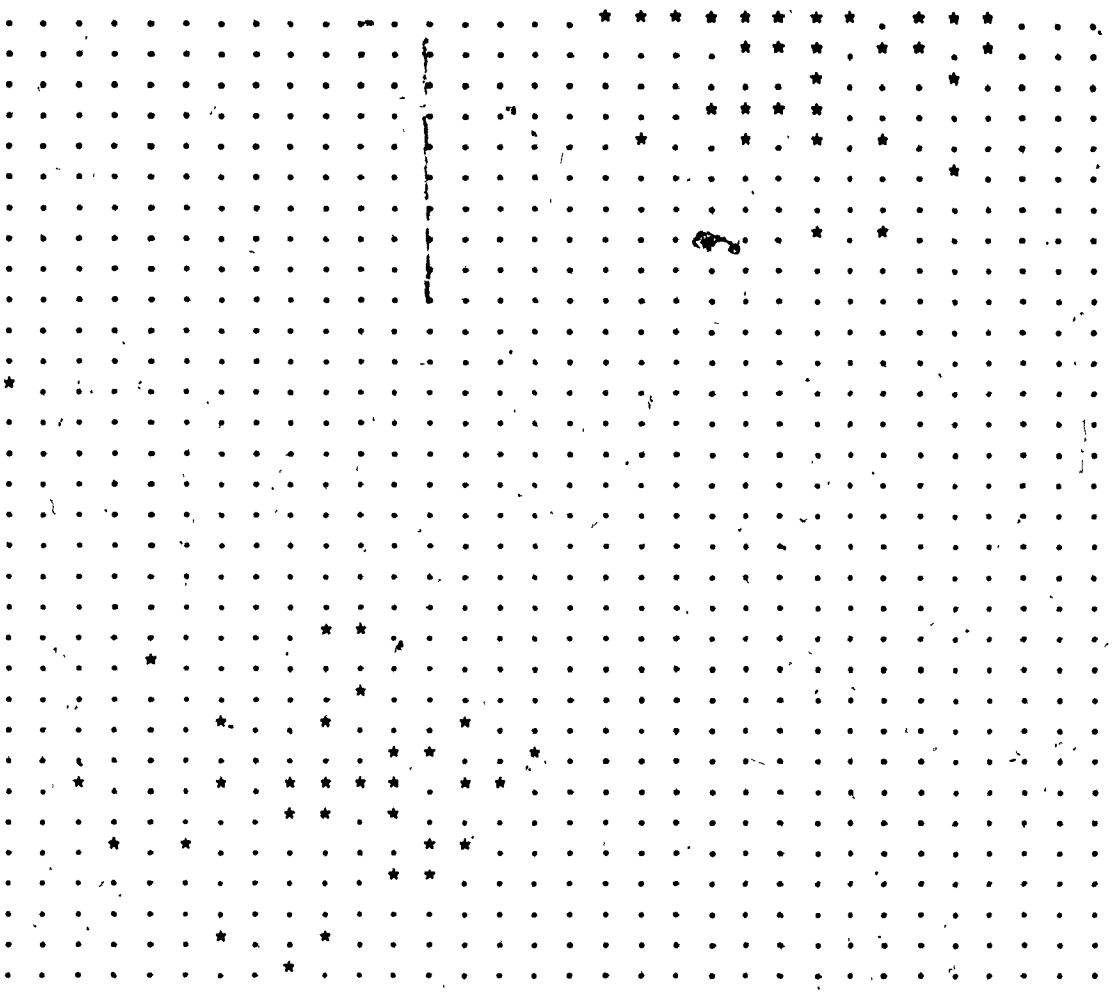
Uniformly distributed faults: number of faults = 30.



Clustered faults: number of faults = 30; cluster size = 30.



Uniformly distributed faults: number of faults = 61.



Clustered faults: number of faults = 61; cluster size = 31.