Computational Arrays with Flexible Redundancy

John Ramirez and Rami Melhem, *Member, IEEE*

Abstract— **Different multiple redundancy schemes for fault detection and correction in computational arrays are proposed and analyzed. The basic idea is to embed a logical array of nodes onto a processor/switch array such that d processors, $1 \leq d \leq 4$, are dedicated to the computation associated with each node. The input to a node is directed to the d processors constituting that node, and the output of the node is computed by taking a majority vote among the outputs of the d processors. The proposed processor/switch array (PSVA) is versatile in the sense that it may be configured as a nonredundant system or as a system which supports double, triple or quadruple redundancy. It also allows for spares to be distributed in the PSVA in a way that permits spare sharing among nodes, thus enhancing the overall system reliability.**

In addition to choosing the required degree of redundancy, the flexibility of the PSVA architecture allows for the embedding of redundant arrays onto defective PSVA's and for run-time reconfiguration to avoid faulty processors and switches. Different embedding and reconfiguration algorithms are presented and analyzed using Markov chain techniques, using probability arguments, and via simulation.

Index Terms—**Multiple redundancy, fault tolerant arrays, reconfiguration, embedding, defect avoidance, fault masking.**

I. Introduction

MOST research in fault tolerant computational arrays has concentrated on defect avoidance and fault coverage (see for e.g., [2], [3], [5], [11], [18], [19], [21]) while only few research efforts have been directed toward fault detection and correction in such arrays. The roving spare technique [20], the weighted checksum coding [4], [8] and overlapping H processes [14] are examples of approaches that may be used to detect (and in some cases correct) transient or permanent run-time faults. However, in these approaches, a latency period may elapse before faulty processors are detected and identified. If the processor array is subject to severe recovery time constraints, or if the production of faulty results may be disastrous, then the use of modular redundancy is appropriate.

Active modular redundancy has been used in highly reliable computing systems (see e.g., [6], [7], [13], [23]) to detect and dynamically mask faults. Recently, Kiskis and Shin [10] suggested a technique for embedding triple modular redundancy into hypercubes by assigning each task to three processors in the hypercube. Their goal is to mask any single fault and yet, retain the logical hypercube connection. In the context of computational arrays, modular redundancy may be achieved by replicating each node in a logical array d times. The input to a node is directed to its d replicas, and the output of the node is computed by taking a majority vote among the outputs of the d replicas. In Section 2 of this paper, we introduce a versatile architecture which is based on interleaved arrays of processors and switches, and which permits the implementation of different degrees of redundancy. For example, two dimensional logical arrays may be embedded onto this architecture with $d = 1$ for no fault tolerance, $d = 2$ for fault detection, $d = 3$ for fault correction/masking and $d = 4$ if additional sparing is required.

The flexibility of the suggested architecture also allows for reconfiguration strategies that may be applied at fabrication time, compile time or at run time. At fabrication or compile time, reconfiguration may be applied to embed a specified logical array onto the architecture in a way that avoids defects or faults, respectively. At run time, reconfiguration may be used to repair the array after faults or to improve the quality of the embedding, thus improving the reliability of the system [16]. In all cases, the new configuration should be valid in the sense that it should allow for each logical node to be appropriately connected to its logical neighbors via the existing physical communication links. Conditions for the validity of configurations are discussed in Section III assuming that processors and switches may fault, but that a faulty switch may be used as a short circuit connection [9], [12].

Two strategies are presented for embedding logical arrays onto defective processor switch/voter arrays (PSVA's for short) with a given redundancy. In Section IV, a greedy strategy is discussed to embed a maximal size logical array onto a given defective PSVA. In this strategy, nodes are embedded sequentially, and each logical node is mapped according to the information available about the mappings of the previous nodes. We show that the problem of embedding a maximal linear array onto a defective PSVA can be solved optimally in linear time. For two-dimensional arrays, finding optimal solutions seems to require exponential time complexity, and thus, we explore linear time algorithms that produce sub-optimal solutions. The second embedding strategy is for mapping a fixed size array onto a defective PSVA. It starts from a fixed embedding and incrementally changes it to avoid defects/faults. This strategy is discussed in Section V. The yield of both strategies, which is the probability of successfully completing the embedding, is analyzed using Markov chain techniques, using probability arguments, and via simulation.

The above two reconfiguration schemes are driven by a centralized controller or a host, and thus are suitable for fabrication-time defect tolerance and compile-time fault tolerance. In Section VI, we consider run-time faults in PSVA's

Manuscript received January 31, 1992; revised June 15, 1992. This work was supported in part under NSF Grant MIP-8911303. This work was presented in part at the 1991 International Conference on Parallel Processing.

The authors are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

IEEE Log Number 9214053.

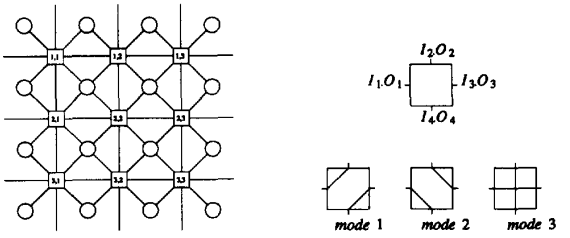


Fig. 1. A processor switch/voter array. (a) A 3 × 3 PSVA. (b) The *pass* modes of a switch. Square = switch/voter, circle = processor.

assuming that a logical array is already embedded onto the PSVA such that at least d processors constitute each logical node. If, at run-time, some elements of the PSVA become faulty, then some logical nodes may be left with less than d processors while some other nodes may have more than d processors. In this case, it may be possible to distributively reconfigure the system at run time to guarantee that each node is d redundant, thus restoring the system's functionality.

In the rest of the paper, it is assumed that communication links are fault free. Reconfiguration algorithms that are more sophisticated than those presented in this paper are needed to reconfigure around faulty links. It is also assumed, as in most multiple redundancy systems, that no two faulty processors produce the same result and that no two processors fault simultaneously at run time. Finally, the analysis and simulation studies are based on random fault distributions in which switch/voter failures are independent of processor failures.

II. A Versatile Architecture for Redundant Arrays

In this section, we introduce the Processor Switch/Voter Array architecture (PSVA). It consists of a mesh-connected array of switch/voters (referred to as switches), where each switch is connected to four processors in the manner indicated in Fig. 1. Conversely, each processor (except border processors) is connected to four switches. Let n_r and n_c be the numbers of rows of switches and columns of switches, respectively, in the PSVA. Then, the number of switches is $n_r \times n_c$, and the number of processors is $(n_r + 1) \times (n_c + 1)$. If, for simplicity, we assume that $n_r = n_c = n_s$, then a switch may be denoted by $switch_{i,j}$, $1 \leq i, j \leq n_s$ and a processor may be denoted by $proc_{i,j}, 0 \leq i, j \leq n_s$.

A connection between a switch and one of the processors associated with it may be *active* or *inactive*. A switch may have up to four active connections at a time, but a processor may have only one. The *active set*, $C_{i,j}$, for $switch_{i,j}$ is defined to be the set of processors with active connections to that switch. Let $c_{i,j}$ denote the cardinality of $C_{i,j}$. Clearly, for any $i, j : 0 \leq c_{i,j} \leq 4$.

A. Operation Modes of Switches

When a switch receives a data item from a neighboring switch, it replicates the item $c_{i,j}$ times and passes a copy along each active connection to the processors. Data items received by a switch from processors in the switch's active set, however, are treated differently depending upon the value of $c_{i,j}$. Specifically, the value of $c_{i,j}$ specifies the mode of the switch as follows.

1) If $c_{i,j} = 0$, $switch_{i,j}$ has no active links to any processors. In this case, the switch is said to be in *pass* mode. A switch/voter in pass mode functions only as a switch, and it directly connects its inputs and outputs in one of the ways depicted in Fig. 1(b).
2) If $c_{i,j} = 1$ the switch is said to be in *straight* mode and data received by the switch from the processor is simply forwarded to the destination switch.
3) If $c_{i,j} = 2$ the switch is said to be in *checking* mode and data received from the two processors are compared. If they match, a copy is sent to the destination; otherwise an error flag is set.
4) If $c_{i,j} > 2$ the switch is said to be in *voting* mode. In this case, the result of a majority vote on the data from the processors is forwarded to the destination. The data that disagree with the majority are deemed faulty, and the processor that sent them is removed from $C_{i,j}$.

B. Switch Design

Assume that $I_k, O_k, k = 1, \cdots, 4$, are the input and output links, respectively, connecting a switch to its four neighboring switches (see Fig. 1). To implement the different switch modes, input from a given I_k should be sent to the processors in $C_{i,j}$ along the appropriate links. Data sent back to the switch are voted on and the result is sent to its proper destination output O_k. Given that there are four possible destinations; $O_1(west), O_2(north), O_3(east)$ and $O_4(south)$, the correct destination of the data may be determined by using one of two methods. Namely, 1) Two bits may be added to the data to determine the direction, or 2) Message cycles may be divided into 4 sub-cycles, one for each direction. We will first present a design for the switch/voters that assumes that the second method is used, and then point to the modifications required to accommodate the first method.

Fig. 2 shows a possible design for a bit-serial switch/voter when each message cycle is divided into four sub-cycles that are globally synchronized. The status of the switch is controlled by a status word, $SW = sw_1, \cdots, sw_6$. The two bits sw_5 and sw_6 control the *pass* modes; 01, 10 and 11, represent the three *pass* modes of Fig. 1(b), while 00 means that the switch is not in the *pass* mode. These two bits generate the signals $x_k, k = 1, \cdots, 4$ that control a set of 2-1 multiplexers and demultiplexers for connecting the appropriate inputs and outputs thus bypassing the switch/voter (see the bottom of Fig. 2). The four other bits in SW specify a particular *nonpass* switch mode. Assuming that the four processors connected to the switch are labeled by 1, 2, 3, and 4, then $sw_m = 1$ iff processor m is actively connected to the switch. The connections to and from processor m are labeled in Fig. 2 by in_m and o_m, respectively.

The multiplexer, mux, in Fig. 2, samples I_k, $k = 1, \cdots, 4$, in four consecutive sub-cycles and sends the multiplexed data to the active processors. Connections to nonactive processors are masked through three-state, high impedance, gates

TABLE I
VOTING BY TABLE LOOKUP

Inputs								Outputs					
o_1	sw_1	o_2	sw_2	o_3	sw_3	o_4	sw_4	Maj	sw_1	sw_2	sw_3	sw_4	error
0	1	0	1	0	1	0	1	0	1	1	1	1	0
0	1	0	1	1	1	0	1	0	1	1	0	1	0
1	1	1	1	–	0	1	1	1	1	1	0	1	0
0	1	0	1	–	0	1	1	0	1	1	0	0	0
1	1	0	1	–	0	–	0	1	1	1	0	0	1
0	1	–	0	–	0	–	0	0	1	0	0	0	0

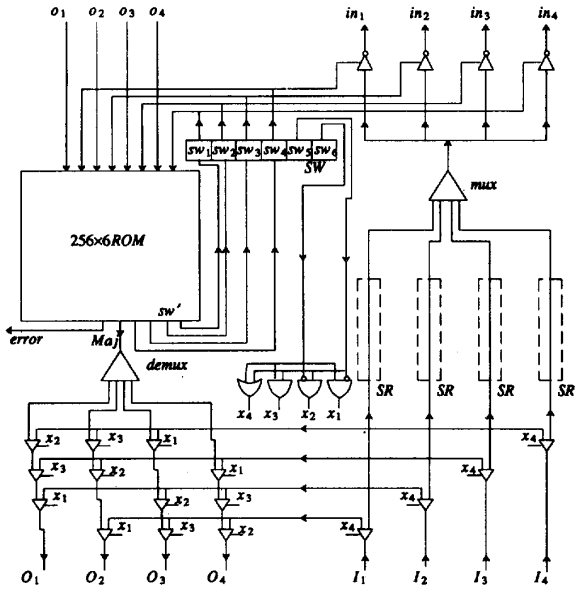


Fig. 2. Schematic diagram of a switch/voter.

controlled by $sw_k, k = 1, \cdots, 4$. These four bits are also combined with the outputs from the processors (received on $o_k, k = 1, \cdots, 4$) to determine the majority Maj, which is then demultiplexed to $O_1, \cdots, O_4$ in every four consecutive sub-cycles. The voting as well as the exclusion of faulty processors from the active set of processors is done via a Read Only Memory. For any combination of active processors and input data (there are 2^8 combinations), the ROM stores the majority, Maj, an $error$ bit that is set when no majority can be obtained, and a new set of active processors determined by four bits, $sw'_k, k = 1, \cdots, 4$, that are used to over-write $sw_k, k = 1, \cdots, 4$. Specifically, the ROM stores the functions

$$Maj = \begin{cases} \text{majority } \{o_k; sw_k = 1\}, & \text{if error } \neq 1, \\ \text{don't care}, & \text{otherwise}, \end{cases}$$

$$sw'_k = (o_k \equiv Maj) \text{ and } sw_k.$$

That is processor k is kept in the active set only if it was active ($sw_k = 1$) and o_k agrees with the majority. In Table I, we show a few entries (out of order) of the ROM. The first shown entry represents four active nonfaulty processors, while the second entry represents four active processors with processor 3 being faulty. The third and fourth entries represent three active processors, the fifth entry represents two active processors and the last entry represents only one active processor. A dash in the table represents a don't care.

If the communication sub-cycles are not globally synchronized, then messages should be appropriately framed and buffers should be added at the input links (example, SR on links I_k, $k = 1, \cdots, 4$, in Fig. 2). Furthermore, if instead of dividing a communication cycle into four sub-cycles, two bits are used to determine the direction of a message, then mux and $demux$ in Fig. 2 should be replaced by a message queue handler and a simple 1–4 router, respectively.

It should be noted that the dynamic modification of the SW register is only needed if the active set of processors connected to a switch is to be modified at run-time. Otherwise, a simple voting scheme is needed and the size of the ROM is reduced to 256×1. With this, it is possible to implement the circuit in Fig. 2 using less than 500 transistors. Given that the number of switch/voters is approximately equal to the number of processors in the PSVA, it is only beneficial to implement this multiple redundancy scheme if the complexity of a processor far exceeds the complexity of the circuit in Fig. 2. Moreover, it is not reasonable to assume that switch/voters do not fail. So, we will assume that switches will fail with a probability $(1 - p_s)$, which is a fraction of the probability of a processor failure, $(1 - p)$. We will assume, however, that faulty switches, as well as nonfaulty switches, can be reset to any of the pass modes of Fig. 1(b).

In the given design, the mode of a switch/voter is controlled by setting a 6-bit status word. For fabrication-time and compile-time reconfiguration, it is sufficient to provide a means for a centralized agent/host to set the status word of all the switches in the PSVA. For run-time fault tolerance, however, distributed reconfiguration may require that the status word of a switch be accessible by the active processors connected to it. In Section VI, we discuss an algorithm in which processors need to read and write to the status word of a switch to which they are connected. Problems resulting from the simultaneous writing into the status word of a switch by its active processors may be avoided if the written word is forced through the voting process.

Finally, we note that PSVA's used for compile-time and run-time fault tolerance may utilize relatively elaborate switches since the processors can be complex (the iWarp [17], the Intel Paragon [1] and the connection machine, CM5, are all arrays of complex processors). Fabrication-time reconfiguration, on the other hand, is usually used for array processors that are laid-out on single wafers. This implies relatively simple processors

and thus requires the switches to be as simple as possible. The basic switch/voter described in this section (with 256×1 ROM) is sufficient in this case since global synchronization may be realized and centralized control may be assumed.

C. Array Embedding with Selective Redundancy

The embedding of an $(n_x \times n_y)$-node logical array onto a PSVA may be specified in terms of a mapping function $map(i,j) = (map_x(i,j), map_y(i,j))$, that maps each node (i,j) to $switch_{map(i,j)}$, and by the sets $C_{map(i,j)}$ that specify the nonfaulty processors that are actively connected to each $switch_{map(i,j)}$ to constitute logical node (i,j). In such an embedding, a logical node, (i,j), is replicated $c_{map(i,j)}$ times in the PSVA, and thus, by choosing the appropriate embedding, a given degree of redundancy may be obtained. For example, $map(i,j) = (2i-1, j)$, $i = 1, \cdots, (n_s+1)/2$, $j = 1, \cdots, n_s$, along with $C_{2i-1,j} = \{proc_{2i-2,j-1}, proc_{2i-1,j-1}\}$ represent the embedding of an $(n_s+1)/2 \times n_s$ logical array into an $n_s \times n_s$ PSVA with double redundancy (see Fig. 3(b), where only active connections are shown). Similarly, triply redundant arrays are obtained by the mapping (see Fig. 3(c)):

$$map(i,j) = \left(2i-1, j + \left\lfloor \frac{(j-1)}{2} \right\rfloor\right)$$
$$i = 1, \cdots, \left\lfloor \frac{n_s+1}{2} \right\rfloor \text{ and } j = 1, \cdots, 2\left\lfloor \frac{n_s+1}{3} \right\rfloor$$

and $C_{map(i,j)} = \{proc_{map(i,j)}, proc_{map(i,j)-(1,1)}, proc_{map(i,j)+\delta(j)}\}$, where $\delta(j) = (-1,0)$ or $(0,-1)$ if j is even or odd, respectively, and the addition operation is extended to tuples such that $(u,v)+(w,z) = (u+w, v+z)$. Finally, for quadruple redundancy (Fig. 3(d)), the mapping is

$$map(i,j) = (2i-1, 2j-1)\ i,j = 1, \cdots, \left\lfloor \frac{n_s+1}{2} \right\rfloor \quad (1)$$

and $C_{map(i,j)} = \{proc_{(map(i,j)+\delta)}; \delta = (0,0), (0,-1), (-1,0)$ and $(-1,-1)\}$. In any of the above mappings, any switch that is not used to map a node is set to *pass* mode 3.

The reliability analysis for PSVA arrays is straight forward and depends on the definition of system failure. For instance, consider the mapping described by (1) and assume that the PSVA fails when at least one node loses its fault correction capability. That is, when the array may no longer mask any single fault. In this case, failure occurs if, for some (i,j), either $switch_{map(i,j)}$ fails or more than one processor connected to that switch fails. The system's reliability is thus:

$$R_3 = (p^4 + 4(1-p)p^3)^N p_s^N$$

where p and p_s are the reliabilities of a processor and a switch, respectively, and $N = \lfloor (n_s+1)/2 \rfloor^2$, is the total number of logical nodes mapped to the PSVA. On the other hand, if the system is not considered to have failed until the detection of a fault that cannot be corrected, then failure requires that more than two processors connected to an active switch fail. That is, the reliability of the system is:

$$R_2 = (p^4 + 4(1-p)p^3 + 6(1-p)^2p^2)^N p_s^N.$$

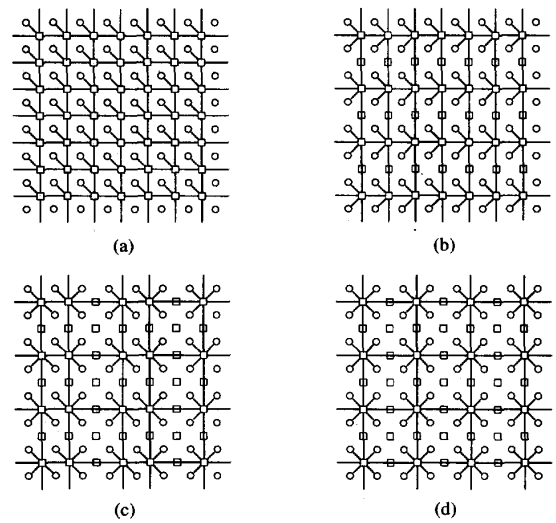


Fig. 3. Embedding two-dimensional arrays onto a 7×7 PSVA. (a) 7×7 array ($d = 1$). (b) 4×7 array ($d = 2$). (c) 4×5 array ($d = 3$). (4) 4×4 array ($d = 4$).

In addition to choosing the required degree of redundancy in PSVA's, the flexible *pass* modes of the switches offer the capability of constructing redundant arrays even in the presence of defective processors and switches. This is demonstrated next.

III. Valid Mappings of Logical Arrays onto PSVA's

Given a PSVA and assuming that some processors and switches in the array are defective, then it is possible to embed a logical array with a given redundancy onto the defective PSVA if defective switches can operate correctly when set to the *pass* mode. The embedding problem is formally specified as one of finding a function, *map*, and the corresponding active sets, C_{map}, such that:

1) each node (i,j) in the logical array is mapped to a nondefective switch, $switch_{map(i,j)}$, in a way that allows for connections to be established between the *north*, *south*, *east* and *west* ports of each $switch_{map(i,j)}$ and, respectively, the *south* port of $switch_{map(i-1,j)}$, the *north* port of $switch_{map(i+1,j)}$, the *west* port of $switch_{map(i,j+1)}$ and the *east* port of $switch_{map(i,j-1)}$;
2) each $switch_{map(i,j)}$ is connected to at least d nondefective processors which are assigned to the active set $C_{map(i,j)}$. The active sets should satisfy $C_{map(i,j)} \cap C_{map(i',j')} = \emptyset$, for any $(i,j) \neq (i',j')$.

A mapping function which satisfies the first condition with single channel communication links is called a *valid* mapping. That is, a valid mapping allows for the mesh connections between logical nodes to be established assuming single duplex channels between adjacent switches. Necessary and sufficient conditions for valid mappings are presented and analyzed in [15]. These conditions are derived for general mappings and are rather complex. In this paper, we will restrict ourselves to two specific classes of mapping functions. Namely, *row-*

wise mappings and $shift\_1$ mappings. For these two classes, validity conditions are straight forward to verify.

In a *row-wise* mapping, any row of logical nodes is mapped to the same row of switches. That is $map_x(i,j) = map_x(i,j-1)$ and $map_y(i,j) > map_y(i,j-1)$, for any i and any $j > 1$. We further restrict *row-wise* mappings such that the first row of logical nodes is mapped to the first row of switches, and consecutive rows of logical nodes are mapped to alternate rows of switches. That is $map_x(i,j) = 2i-1$. With these restrictions, we can prove that the mapping is valid if the column position of any node, (i,j), is somewhere between the column positions of nodes $(i-1,j-1)$ and $(i-1,j+1)$. Specifically, we can prove the following:

Proposition 1: A row-wise mapping, map, which satisfies $map_x(i,j) = 2i-1$ is valid if, for each i and j,

$$map_y(i-1,j-1) \leq map_y(i,j) \leq map_y(i-1,j+1)$$

Proof: Consider the case $map_y(i,j) \leq map_y(i-1,j)$ (the case $map_y(i,j) > map_y(i-1,j)$ may be proved using a similar argument). The horizontal connections are always possible in a row-wise mapping. The connection between the *south* port of a switch, $switch_{map(i-1,j)}$, and the *north* port of $switch_{map(i,j)}$ is established through $switch_{2i-2,k}$, $k = map_y(i,j), \cdots, map_y(i-1,j)$. It is easy to see that all these switches, except possibly $switch_{2i-2,map_y(i,j)}$ and $switch_{2i-2,map_y(i-1,j)}$, are not used for any other connections. The worst case is when $switch_{2i-2,map_y(i,j)}$ is used to connect $switch_{map(i-1,j-1)}$ to $switch_{map(i,j-1)}$ and $switch_{2i-2,map_y(i-1,j)}$ is used to connect $switch_{map(i-1,j+1)}$ to $switch_{map(i,j+1)}$. As seen from Fig. 4(a), each of the two switches can support the required connections. □

In a Shift_1 mapping, an initial fixed valid mapping, $fmap$ is given, and the mapping is obtained by possibly shifting each node from its position in $fmap$ by at most one switch in one of the four normal directions. That is, $map(i,j) = fmap(i,j) + \Delta(i,j)$, where $\Delta(i,j) = (0,0),(1,0),(0,1),(-1,0)$ or $(0,-1)$. Note that even if $fmap$ is a valid mapping, the Shift_1 version of that mapping may not be a valid mapping. The following proposition establishes the validity of the Shift_1 mapping in the case where the initial mapping is $fmap(i,j) = (2i-1, 2j-1)$. Specifically, for the mapping to be valid, any two horizontally adjacent nodes that are mapped to different rows of switches should be separated by at least one column of switches. A similar condition applies to vertically adjacent nodes. This result will be used in Section 5.

Proposition 2: The mapping, $map(i,j) = (2i-1, 2j-1) + \Delta(i,j)$, where $\Delta(i,j) = (0,0)$, $(\pm 1, 0)$ or $(0, \pm 1)$, is valid if, for any i and j, the following two conditions are satisfied:

a) either $map_x(i,j) = map_x(i,j-1)$ or $map_y(i,j) - map_y(i,j-1) > 1$
b) either $map_y(i,j) = map_y(i-1,j)$ or $map_x(i,j) - map_x(i-1,j) > 1$

Proof: We prove that the second condition always allows for a vertical connection between $switch_{map(i,j)}$ and $switch_{map(i-1,j)}$. Let $d_y = map_y(i,j) - map_y(i-1,j)$ and $d_x = map_x(i,j) - map_x(i-1,j)$. For $d_y = 0$, the vertical connections may be easily made. Moreover, for $d_y \neq 0$, the given restrictions on the values of Δ restrict the values of (d_x, d_y) to one of the following: $(1,\pm 1)$, $(2,\pm 1)$, $(3,\pm 1)$ and $(2,\pm 2)$. In Fig. 4(b), it is shown that for the cases with $d_x > 1$, the vertical connection may always be established. Due to symmetry, only the cases with positive d_y are shown. Similarly, we may define $\bar{d}_y = map_y(i,j) - map_y(i,j-1)$ and $\bar{d}_x = map_x(i,j) - map_x(i,j-1)$ and prove that the first condition allows for the horizontal connections between $switch_{map(i,j)}$ and $switch_{map(i,j-1)}$. Moreover, by considering possible combinations of (d_x, d_y) and $(\bar{d}_x, \bar{d}_y)$, it may be shown that the vertical connections never interfere with the horizontal connections. □

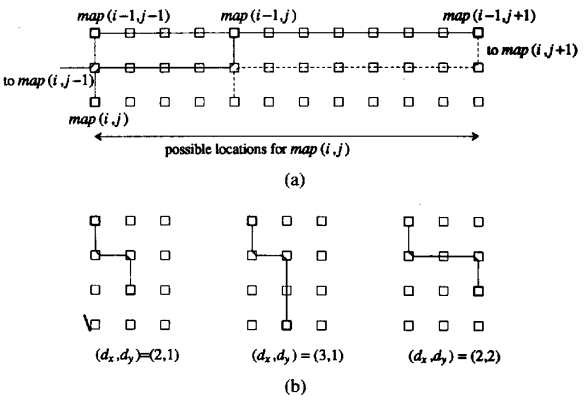


Fig. 4. Establishing the vertical connections in valid mappings.

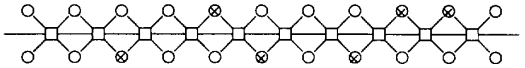


Fig. 5. A linear PSVA with $n_s = 10$ switches.

In the following two sections, we discuss two policies for mapping a logical array to a defective PSVA with a given redundancy.

IV. The Greedy Mapping of Logical Arrays onto Defective PSVA's

The goal of the greedy policies described here is to map a maximal size array onto a given defective PSVA. We start by presenting a greedy algorithm for embedding a maximal linear logical array onto a defective linear PSVA. In this context a linear PSVA is one that consists of one row of switches and two rows of processors as shown in Fig. 5. In this case, we will denote $switch_{1,j}$ by $switch_j$ and $map(1,j)$ by $map(j)$.

A. Greedy Embeddings for Linear Arrays

Given a linear PSVA with defective switches and processors, the goal is to embed a maximal size logical array onto the PSVA such that each node in the array is d-redundant for a given d, $d \leq 4$. The algorithm visits the switches sequentially from left to right, and tries to embed a logical node at each visited switch. It assumes that all the defective processors are initially *marked* to indicate that they are *unavailable*. Also,

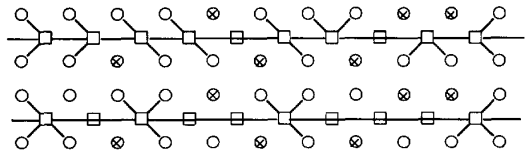


Fig. 6. Greedy embeddings onto the array of Fig. 5 with $d = 2$ and $d = 3$.

the predicate $OK(switch)$ is used to indicate that $switch$ is not defective.

Algorithm L_Greedy:

1) *logical_node = 0 ;*
2) FOR $(k = 1; k \leq n_s; k++)$
   IF $OK(switch_k)$ THEN
   2.1) Let H_k=set of un-marked processors connected to $switch_k$;
   2.2) IF $|H_k| \geq d$ then
      2.2.1) Let *logical_node* =*logical_node*+1 and *map (logical_node)* = k;
      2.2.2) Let C_k be a subset of H_k that contains exactly d processors. Give the processors $proc_{0,k-1}$ and $proc_{1,k-1}$ a higher priority for inclusion in C_k ;
      2.2.3) Mark the processors in C_k *unavailable;*
3)n_g= *logical_node* ;/* the size of the array successfully embedded in the PSVA */

For example, in Fig. 6, we show the greedy embedding of an 8-node array and a 4-node array with $d = 2$ and 3, respectively, onto the defective PSVA of Fig. 5. Although greedy algorithms are not usually optimal, it may be shown that the above greedy strategy is optimal.

Proposition 3: L_Greedy is optimal in the sense that no other algorithm can embed an array with more than n_g nodes in the PSVA.

Proof: Define $R(j)$ as the number of nondefective processors among $\{proc_{0,map(j)}, proc_{1,map(j)}\}$ that are not marked after logical node j is mapped onto $switch_{map(j)}$. Clearly, $R(j) = 0$, 1 or 2, and if $d > 1$, then $R(j) = |H_{map(j)}| - |C_{map(j)}|$. Also, let map_{opt} be an optimal mapping which maps node j to $switch_{map_{\text{opt}}(j)}$, and $R_{\text{opt}}(j)$ be the number of nondefective processors among $\{proc_{0,map_{\text{opt}}(j)}, proc_{1,map_{\text{opt}}(j)}\}$ that are not used for node j. Given that $map(1)$ is the first nondefective switch that is connected to at least d nondefective processors and that step 2.2.2 of L_Greedy maximizes $R(1)$, then either $map(1) < map_{\text{opt}}(1)$ or $[map(1) = map_{\text{opt}}(1)$ and $R(1) \geq R_{\text{opt}}(1)]$. Now, assume that $map(j-1) < map_{\text{opt}}(j-1)$ or $[map(j-1) = map_{\text{opt}}(j-1)$ and $R(j-1) \geq R_{\text{opt}}(j-1)]$. In either case, it is easy to see that L_Greedy ensures that $map(j) < map_{\text{opt}}(j)$ or that $[map(j) = map_{\text{opt}}(j)$ and $R(j) \geq R_{\text{opt}}(j)]$. This induction shows that if an optimal algorithm may embed an array with n nodes onto a PSVA, then $map(n) \leq map_{\text{opt}}(n)$ and thus L_Greedy will also successfully embed such an array onto the PSVA. □

The yield, $Y_{L\_Greedy}(d, n_a, n_s)$ is the probability of successfully embedding a logical array with n_a or more nodes onto a defective n_s-switch PSVA using L_Greedy. Using a layered Markov chain, this yield may be computed assuming a given probability, p, that a processor is not defective and a given probability, p_s, that a switch is not defective. To illustrate the technique, we analyze the case $d = 3$ by considering a Markov chain which consists of $n_a + 1$ layers, $y = 0, 1, \cdots, n_a$, each containing three states, $s_{y,0}$, $s_{y,1}$ and $s_{y,2}$ (see Fig. 7). A transition in the Markov process corresponds to the execution of one iteration of step 2 in L_Greedy. The semantics of the states is such that, after k transitions, the Markov process is in state $s_{y,u}$ if, after k iterations of L_Greedy, y nodes are successfully mapped to some switches in $\{switch_1, \cdots, switch_k\}$ and u processors among $\{proc_{0,k}, proc_{1,k}\}$ are not marked. Thus, these u processors are available to be actively connected to $switch_{k+1}$.

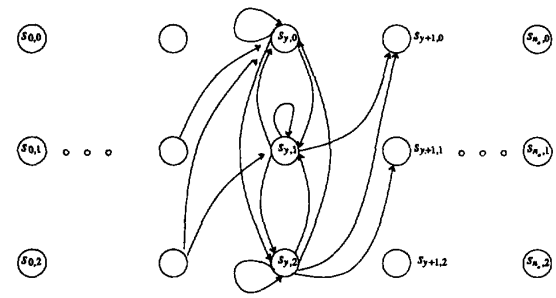


Fig. 7. A Markov chain for L_Greedy (only transitions to and from $S_{y,u}, u = 0, 1, 2$, are shown).

According to the above semantics, if the kth transition is between two states in a given layer, y, then this indicates that iteration k fails to map node $y+1$. On the other hand, if the kth transition is from a state in layer y to a state in layer $y+1$, then this indicates that iteration k succeeds in mapping node $y+1$. The probabilities of such transitions may be obtained in terms of p and p_s. Specifically, if after k transitions, the process is in state $s_{y,u}$, then the different transition probabilities out of $s_{y,u}$ will depend on the status (defective or not) of $switch_{k+1}$, $proc_{0,k+1}$ and $proc_{1,k+1}$.

For example, assume that after k transitions the process is in $s_{y,1}$. That is, only one of $proc_{0,k}$ or $proc_{1,k}$ is available to be actively connected to $switch_{k+1}$ (Fig. 8(a)). Hence, mapping node $y+1$ to $switch_{k+1}$ is only possible if $proc_{0,k+1}$, $proc_{1,k+1}$ and $switch_{k+1}$ are not defective (with probability $p^2 p_s$). In this case, a transition to state $s_{y+1,0}$ occurs (Fig. 8(b)). If one of $proc_{0,k+1}$ or $proc_{1,k+1}$ is defective (with probability $2p(1-p)$), then the process remains in state $s_{y,1}$ (Fig. 8(c)). If both $proc_{0,k+1}$ and $proc_{1,k+1}$ are defective (with probability $(1-p)^2$), a transition to state $s_{y,0}$ occurs (Fig. 8(d)). Finally, if $switch_{k+1}$ is defective and both $proc_{0,k+1}$ and $proc_{1,k+1}$ are not defective (with probability $(1-p_s)p^2$), a transition to state $s_{y,2}$ occurs (Fig. 8(e)). Note that the above 4 cases represent all possible defect configurations for $switch_{k+1}$, $proc_{0,k+1}$ and $proc_{1,k+1}$.

The transition probabilities out of $s_{y,0}$ and $s_{y,2}$ may be calculated in a similar manner. The results are given by the following equation in which $\sigma^k_{y,u}$ is the probability of being

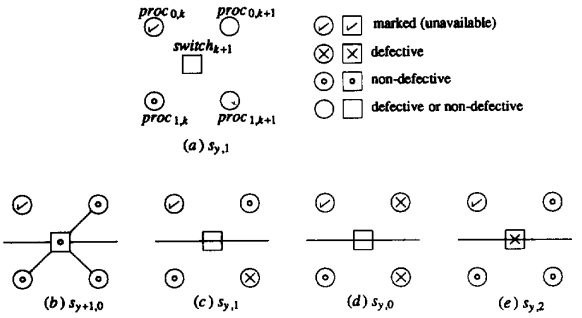


Fig. 8. (a) State before iteration k, (b), (c), (d), (e) states after iteration k.

in state $s_{y,u}$ after the kth transition:

$$\begin{bmatrix}\sigma_{y,0}^{k+1}\\ \sigma_{y,1}^{k+1}\\ \sigma_{y,2}^{k+1}\end{bmatrix} = \begin{bmatrix}(1-p)^2 & (1-p)^2 & (1-p)^2\\ 2p(1-p) & 2p(1-p) & 2p(1-p)(1-p_s)\\ p^2 & p^2(1-p_s) & p^2(1-p_s)\end{bmatrix} \cdot \begin{bmatrix}\sigma_{y,0}^{k}\\ \sigma_{y,1}^{k}\\ \sigma_{y,2}^{k}\end{bmatrix} + \begin{bmatrix}0 & p^2p_s & 2p(1-p)p_s\\ 0 & 0 & p^2p_s\\ 0 & 0 & 0\end{bmatrix} \begin{bmatrix}\sigma_{y-1,0}^{k}\\ \sigma_{y-1,1}^{k}\\ \sigma_{y-1,2}^{k}\end{bmatrix}$$

If we denote the first matrix by A and the second matrix by B, then the single transition probabilities in the entire Markov process is:

$$\sigma^{k+1} = M\sigma^k = \begin{bmatrix}A & O & . & . & O & O\\ B & A & . & . & . & .\\ O & B & . & . & . & .\\ . & O & . & . & O & O\\ . & . & . & . & A & O\\ O & . & . & . & B & I\end{bmatrix}\sigma^k$$

where O is a zero matrix, I is a unit matrix, and σ^k is a vector that contains the $3(n_a+1)$ probabilities $\sigma_{y,u}^k$, $y = 0, \cdots, n_a$, $u = 0, 1, 2$. The Markov process starts from $s_{0,0}$, $s_{0,1}$ or $s_{0,2}$ with probabilities $(1-p)^2$, $2p(1-p)$ and p^2, respectively. Hence, all the components of σ^0 are set to zeroes except $\sigma_{0,0}^0$, $\sigma_{0,1}^0$ and $\sigma_{0,2}^0$, which are set to $(1-p)^2$, $2p(1-p)$ and p^2, respectively. The state probabilities after n_s iterations are given by

$$\sigma^{ns} = M^{ns}\sigma^0$$

Noting that states $s_{na,u}$, $u = 0, 1, 2$, are absorbing states because they are reached when L_Greedy successfully embeds n_a nodes in the PSVA, we may compute the yield as: .

$$Y_{L\_Greedy}(3, n_a, n_s) = \sigma_{na,0}^{ns} + \sigma_{na,1}^{ns} + \sigma_{na,2}^{ns}$$

In Fig. 9, we show the result of the above analysis for $n_s = 19$. Namely, we plot the yield for different values of n_a and p assuming that $(1-p_s) = 0.1(1-p)$. Note that if all processors and switches are nondefective, then a 13-node logical array may be embedded in the 19-switch linear PSVA.

B. Row-Wise Greedy Embeddings for Two-Dimensional Arrays

Given a defective $n_s \times n_s$ PSVA, the goal is to embed a maximal $n_x \times n_y$ logical array onto the PSVA. The greedy algorithm for linear arrays may be extended to a row-wise

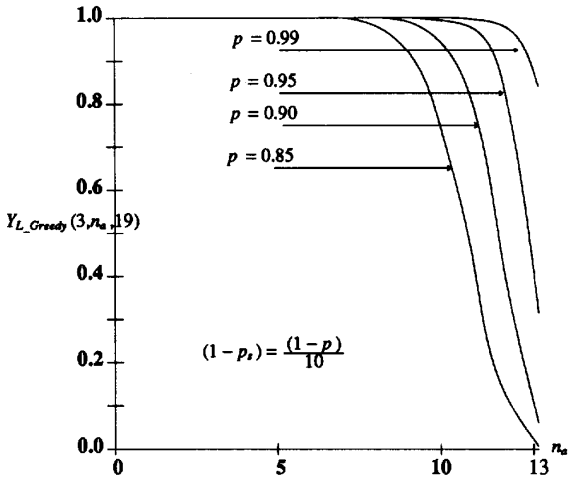


Fig. 9. The probability of embedding n_a nodes onto a 19-switch PSVA.

greedy algorithm for two dimensional arrays. Specifically, we assume that row i of the logical array is mapped to row $2i-1$ of switches using the linear greedy algorithm. This fixes the number of rows in the logical array to $n_l = \lfloor (n_s+1)/2 \rfloor$, and the goal becomes to maximize the number of columns n_y in the logical array. However, according to Proposition 1, the mapping of row i, $i > 1$, is not independent from the mapping of row $i-1$. Moreover, in order to end up with a regular logical array, each logical row should contain the same number of logical nodes. Taking this into consideration, the following greedy algorithm may be used to map a logical $n_l \times n_y$ array onto a defective PSVA with the objective of maximizing n_y. The algorithm assumes that all defective processors in the PSVA are initially marked *unavailable*.

Algorithm 2-D_Greedy

Use L_Greedy to embed logical row 1 to the first row of switches. Let n_y be the number of nodes resulting from that embedding.

FOR $(i = 2; i \le \lfloor \frac{n_s+1}{2} \rfloor; i++)$

FOR $(j = 0; j < n_y;)$

1) FOR (k =max $\{map_y(i,j)+1, map_y(i-1,j)\}$; $k \le$ min $\{n_s, map_y(i-1, j+2)\}$; $k++$)

IF *OK* $(switch_{2i-1,k})$ THEN

1.1) Let $H_{2i-1,k}$ = set of un-marked processors connected to $switch_{2i-1,k}$;

1.2) IF $|H_{2i-1,k}| \ge d$ THEN

1.2.1) Let $j = j+1$ and $map(i,j) = (2i-1, k)$;

1.2.2) Let $C_{2i-1,k}$ be a subset of $H_{2i-1,k}$ that contains exactly d processors. Give $proc_{2i-2,k-1}$ and $proc_{2i-1,k-1}$ higher priority for inclusion in $C_{2i-1,k}$.

1.2.3) Mark the processors in $C_{2i-1,k}$ *unavailable*, and exit the inner most For loop;

2) IF node $(i, j+1)$ is not mapped in step 1, THEN set $map(l,m) = map(l, m+1)$ for $l = 1, \cdots, i-1$, $m = j+1, \cdots, n_y - 1$ and set $n_y = n_y - 1$;

3) IF $k = n_s$ and $j < n_y$, THEN $n_y = j$

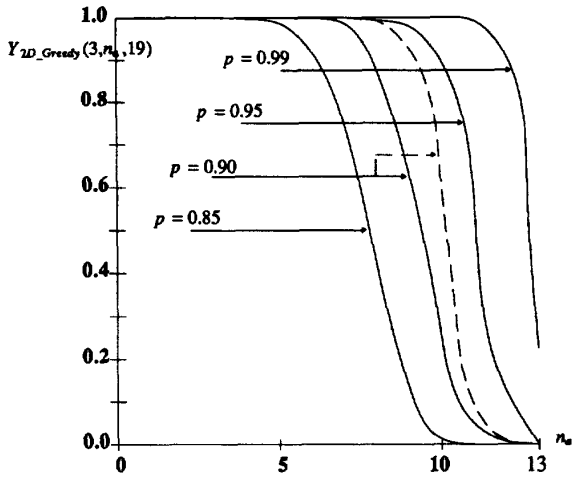


Fig. 10. The yield of 2-D_*Greedy*.

In this algorithm, the condition of Proposition 1 is implicitly imposed in the bounds of the loop in Step 1. After embedding n_y nodes in each of rows $1, \cdots, i-1$, if, in row i, a node (i, j) cannot be embedded, then the entire jth column is removed from the embedding of the previous rows (Step 2). Similarly, if in row i, only j nodes, $j < n_y$, can be embedded, then columns $j+1, \cdots, n_y$ are deleted from the previous rows (Step 3). With this simple policy, the running time of the algorithm is $O(n_s^2)$. More complex algorithms that involve back-tracking may be designed (no longer greedy). In such algorithms, previously embedded rows would be incrementally updated to accommodate the embedding of a given row.

In Fig. 10, we show simulation results for $Y_{2-D\_Greedy}(3, n_y, 19)$, the probability of successfully embedding a logical array of size at least $n_l \times n_y$ onto a 19×19 PSVA. The dotted curve given for $p = 0.9$ is the yield resulting from completely ignoring the condition of Proposition 1. That is, each of the 10 rows is independently embedded using L_Greedy, and thus the yield is $[Y_{L\_Greedy}(3, n_a, 19)]^{10}$. This represents an upper bound on any 2-D algorithm that uses back-tracking to improve the yield of 2-D_*Greedy*. Given that this upper bound is far from being tight because it ignores completely the conditions of Proposition 1, and by comparing the two curves for $p = 0.9$, our own conclusion is that algorithms more complex than 2-D_Greedy may improve the yield only modestly.

V. Embedding a Fixed Size Array onto a Defective PSVA

A different strategy for embedding logical arrays onto defective PSVA's is to use a valid fixed mapping, $fmap$, which, when used to embed a logical $n_x \times n_y$ array onto a nondefective PSVA, leaves some of the processors unused. Because of these unused processors, the same mapping, $fmap$, may be used to embed a logical $n_x \times n_y$ array onto a defective PSVA.

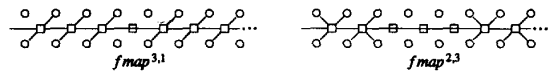


Fig. 11. Initial mapping functions with different utilization factors.

We will consider a specific class of row-wise fixed mappings in which $fmap_x(i, j) = 2i - 1$ and the mapping of each row exhibits some regularity. The mappings will be superscripted in a way that reflects the regularity of row mappings. Specifically, a superscript u, v means that the first u logical nodes in the row are mapped to the first u consecutive switches, then v switches are skipped and the following u nodes are mapped to the following u switches and so on. For example, the mapping of Fig. 3(c) is denoted by $fmap^{2,1}(i, j) = (2i - 1, j + \lfloor (j-1)/2 \rfloor)$, and that of Fig. 3(d) is denoted by $fmap^{1,1}(i, j) = (2i - 1, 2j - 1)$. Other possible fixed mappings are shown in Fig. 11. In this figure, only one row of switches is shown and active connections for $fmap^{3,1}$ and $fmap^{2,3}$ reflect embeddings with $d = 2$ and $d = 3$, respectively.

Given a nondefective $n_s \times n_s$ PSVA and a fixed mapping, $fmap^{u,v}$, the size, $n_x \times n_y$, of the logical array that may be embedded in the PSVA is determined by n_s, u and v. Specifically, $n_x = \lfloor (n_s + 1)/2 \rfloor$ and $n_y = \lfloor (n_s + v)/(u+v) \rfloor u$. Moreover, if a d-redundant embedding that uses $fmap^{u,v}$ utilizes only a fraction of the processors, then this fraction is called the *utilization factor*, $\mu(fmap^{u,v}, d)$. For example, if $fmap^{1,1}(i, j)$ is used with triple redundancy, then the utilization factor is $\mu(fmap^{1,1}, 3) = 0.75$. Similarly, the utilization factors $\mu(fmap^{3,1}, 2) = 0.75$ and $\mu(fmap^{2,3}, 3) = 0.6$.

A utilization factor which is less than unity means that the embedding may still be accomplished if up to $1 - \mu(fmap, d)$ of the processors in the PSVA are defective. This is because, only $\mu(fmap, d)$ of the processors are utilized in the embedding. However, defective switches or uneven distribution of defective processors may not allow for the embedding to be realized with the given mapping $fmap$. In these cases, the shift_1 policy may be useful for obtaining the embedding through a mapping function $map(i, j) = fmap(i, j) + \Delta(i, j)$, where $\Delta(i, j) = (0, 0), (0, \pm 1)$ or $(\pm 1, 0)$.

Hence, assuming that a function *fmap* may embed an $n_x \times n_y$ logical array onto an $n_s \times n_s$ PSVA, with redundancy d and utilization $\mu < 1$, the following algorithm may be used to embed a logical array of the same size onto a defective PSVA, using a shift_1 policy. The algorithm assumes that initially all the defective nodes are marked (to indicate unavailability). The predicate $valid(i, j, \Delta(i, j))$ returns *true* if mapping logical node (i, j) to $switch_{fmap(i,j)+\Delta(i,j)}$ allows this switch to be properly connected to $switch_{map(i,j-1)}$ and $switch_{map(i-1,j)}$, thus guaranteeing a valid mapping.

Algorithm Shift_1/*fmap*:

FOR $i = 1, \cdots, n_x$ do
  FOR $j = 1, \cdots, n_y$ do
    FOR $\Delta(i, j)$ in $\{(0, 0), (-1, 0), (0, -1), (1, 0), (0, 1)\}$ do
      1) $H_{fmap(i,j)+\Delta(i,j)}$ = set of un-marked processors connected to $switch_{fmap(i,j)+\Delta(i,j)}$;

2) IF $OK(switch_{fmap(i,j)+\Delta(i,j)})$ AND $H_{fmap(i,j)+\Delta(i,j)}| \geq d$, AND valid$(i,j,\Delta(i,j))$,
THEN 2.1) $map(i,j) = fmap(i,j) + \Delta(i,j)$
2.2) Mark d processors from $H_{map(i,j)}$ and include them in $C_{map(i,j)}$. Give priority to $proc_{map(i,j)}$, $proc_{map(i,j)+(0,1)}$, $proc_{map(i,j)+(1,0)}$ then $proc_{map(i,j)+(1,1)}$ for inclusion in $C_{map(i,j)}$.
2.3) Exit the inner-most FOR loop.

The above algorithm is considered to have failed if it fails to map any logical node (i,j). Different algorithms may be derived from the above general one for different *fmap* and d. We will analyze only the specific case where $fmap^{1,1}$ is used with $d = 3$. The conditions for the validity of the mapping in this case are given in Proposition 2. Other cases of different *fmap* and d may be analyzed in a similar manner.

The mapping function $fmap^{1,1}$ allows the embedding of an $n_l \times n_l$ logical array, where $n_l = \lfloor (n_s + 1)/2 \rfloor$, onto a nondefective $n_s \times n_s$ PSVA with $d = 3$. The probability that Shift_1/$fmap^{1,1}$ embeds successfully the same logical array onto an $n_s \times n_s$ defective PSVA depends on the defect distribution in the PSVA as well as on the values that are tried for $\Delta(i,j)$ and the order in which these values are tried. In Section 5.2 an algorithm is presented which tries all five values of $\Delta(i,j)$. In the next section, however, the values of $\Delta(i,j)$ are restricted to $(0,0)$, $(0,-1)$ and $(0,1)$.

A. Row-Wise Embedding Using Shift_1/$fmap^{1,1}$

In the row-wise version of Shift_1/$fmap$, each logical node is allowed to be shifted from its position in $fmap^{1,1}$ only in the horizontal direction. That is the values of $\Delta(i,j) = (\pm 1, 0)$ are not tried in algorithm Shift_1/$fmap$. It is straight forward to check that the resulting mapping function, $map = fmap^{1,1} + \Delta$, always satisfies the conditions of Proposition 1. Hence, by decomposing the $n_s \times n_s$ PSVA onto n_l linear PSVA's, the embedding of an $n_l \times n_l$ logical array onto the two dimension PSVA may be achieved by independently embedding each logical row onto the corresponding linear PSVA. The probability of completing the two-dimensional embedding is thus given by

$$Y_{rw\_shift}(3, n_l, n_s) = [Y_{L\_shift}(3, n_l, n_s)]^{n_l} \qquad (2)$$

where, Y_{L_shift}, is the probability of completing the embedding of one logical row onto a linear PSVA using the following algorithm (as in L_Greedy, the row indices are omitted).

Algorithm L_Shift_1/$fmap^{1,1}$:
FOR $j = 1, \cdots, n_l$ do
FOR $\Delta(j) = -1, 0, 1$ do
1) $H_{2j-1+\Delta(j)}$ = set of un-marked processors connected to $switch_{2j-1+\Delta(j)}$;
2) IF $OK(switch_{2j-1+\Delta(j)})$ AND $|H_{2j-1+\Delta(j)}| \geq 3$, THEN
2.1) $map(j) = 2j - 1 + \Delta(j)$
2.2) Mark 3 processors from $H_{map(j)}$ and include them in $C_{map(j)}$. Give priority to $proc_{0,map(j)-1}$ and $proc_{1,map(j)-1}$ for inclusion in $C_{map(j)}$.
2.3) Exit the inner-most FOR loop.

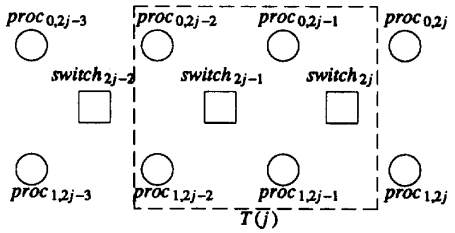


Fig. 12. Processors and switches affecting the embedding of node j.

By trying to embed a logical node, j, with $\Delta(j) = -1$ first, then with $\Delta(j) = 0$ and 1, the algorithm actually compresses the mapping toward the left, thus increasing the probability of completing the embedding. Specifically, if node j can be embedded with either $\Delta(j) = -1$ or $\Delta(j) = 0$, then the embedding with $\Delta(j) = -1$ leaves more processors available for the embedding of node $j+1$ and thus increases the chance of successfully embedding that node. Similar arguments apply to trying $\Delta(j) = 0$ before $\Delta(j) = 1$.

In order to analyze the probability of success of the above algorithm, we observe that in iteration j, the success of mapping logical node j depends on the availability (not defective and not marked) of $switch_{2j-1+\Delta}$, for $\Delta = -1, 0, 1$, and of $proc_{q,2j-l}$, for $q = 0, 1$ and $l = 0, \cdots, 3$ (see Fig. 12). If the embedding of node j is successful, then the availability of processors $proc_{q,2j-l}$, $q, l = 0, 1$, after iteration j, will affect iteration $j + 1$. To express the availability of these processors after iteration j we define the variable $R(j)$ as follows.

- If $\Delta(j) = 0$ or -1, then $R(j)$ is the number of processors among $proc_{0,2j-1}$ and $proc_{1,2j-1}$ that are available (not marked) after iteration j.
- If $\Delta(j) = 1$, then $-R(j)$ is the number of processors among $proc_{0,2j}$ and $proc_{1,2j}$ that are needed to complete the embedding of node j.

In other words, if $R(j)$ is nonnegative, it indicates whether 0, 1 or 2 of $proc_{0,2j-1}$ and $proc_{1,2j-1}$ are available to be used, if needed, in the embedding of logical node $j + 1$. If $R(j)$ is negative, then this indicates that both $proc_{0,2j-1}$ and $proc_{1,2j-1}$ are marked and that $|R(j)|$ processors among $proc_{0,2j}$ and $proc_{1,2j}$ are needed to complete the embedding of node j. That is, node j needs to borrow $|R(j)|$ processors from node $j + 1$.

The progress of the embedding algorithm may be modeled by a Markov process in which the jth transition represents the execution of iteration j of the algorithm. In other words, the state of the process after j transitions reflects the progress of the algorithm after the jth iteration. The Markov process consists of six states with the following semantics (see Fig. 13).

- s_f: indicates that the algorithm has failed in embedding node j. This is an absorbing state because the failure to embed one node means the failure of the algorithm.
- s_0: indicates that $R(j) = 0$ OR ($R(j) > 0$ and $switch_{2j}$ is defective).
- s_u, $u = 1, 2$: indicates that $R(j) = u$ and that $switch_{2j}$ is not defective.
- s_u, $u = -1, -2$: indicates that $R(j) = u$.

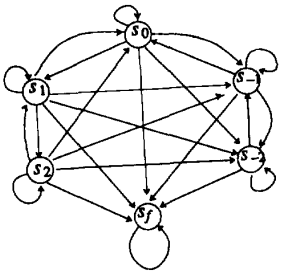


Fig. 13. The Markov chain for L_Shift_1/$fmap^{1,1}$.

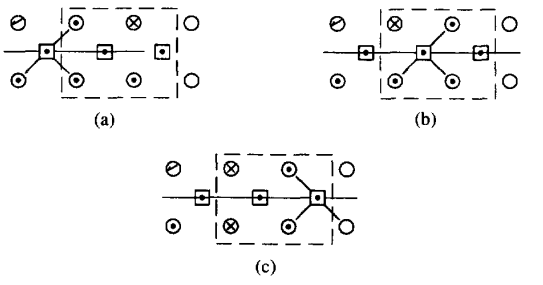


Fig. 14. States after iteration j for three different defect configurations in $T(j)$ assuming that, before iteration j, $proc_{1,2j-3}$ is available and $proc_{0,2j-3}$ is marked (state s_1). (a) s_1. (b) s_0. (c) s_{-1}.

Note that the state information after transition j indicates whether or not $proc_{0,2j-1}$, $proc_{1,2j-1}$ and $switch_{2j}$ are available to be used in embedding node $j+1$ in the next iteration. It also indicates whether $proc_{0,2j}$ and $proc_{1,2j}$ have been used in embedding node j, and thus whether or not they are available to be used for embedding node $j+1$. Two remarks are in order. First, state s_0 specifies that neither $proc_{0,2j-1}$ nor $proc_{1,2j-1}$ can be used for embedding node $j+1$. This is true if $switch_{2j}$ is defective even if $R(j) \neq 0$, since in this case, node $j+1$ cannot be mapped to $switch_{2j}$ and, thus, cannot use $proc_{0,2j-1}$ or $proc_{1,2j-1}$. The second remark is that transitions into state s_{-1} (or s_{-2}) assume that at least one (or both) of $proc_{0,2j}$ and $proc_{1,2j}$ are nondefective.

Let $T(j)$ be the set containing the two switches $switch_{2j-1}$ and $switch_{2j}$, and the four processors connected to $switch_{2j-1}$, and assume that after embedding some node, $j-1$, the Markov process is in state s_u. The probability of successfully embedding node j and the state after that embedding depends on: 1) the state s_u and 2) the probability of different defect configurations in the elements of $T(j)$. Specifically, the transition probabilities out of some state, s_u, may be computed by enumerating all possible configurations of defects in the elements of $T(j)$. There are two switches and four processors in $T(j)$, thus giving 64 defect configurations, each occurring with a given probability expressed in terms of p and p_s. Each configuration will cause a transition to one of the six states in the Markov process and the sum of the probabilities of the configurations leading to a specific state s_v is the transition probability from s_u to s_v.

The above assertion that state transitions depend only on elements in $T(j)$ needs further justification since transitions into s_{-1} (or s_{-2}) assume that at least one (or both) of $proc_{0,2j}$ or $proc_{1,2j}$ is nondefective. Examining the states of these processors during the embedding of node j complicates the Markov analysis since there will not be a clean separation between the elements examined in each iteration. Our solution to this problem is to delay the examination of $proc_{0,2j}$ and $proc_{1,2j}$ until the embedding of node $j+1$. At that time, $proc_{0,2j}$ and $proc_{1,2j}$ are examined as elements of $T(j+1)$ and any incompatibility with the current state will cause a transition to s_f. Note that such incompatibility is only possible if the current state is s_{-1} or s_{-2}. The second example that we discuss below will clarify this point.

For example, in Fig. 14(a), (b), and (c), we assume that after iteration $j-1$, the process is in state s_1 ($proc_{1,2j-3}$ is available—see labels in Fig. 12), and we show three defect configurations for $T(j)$. These configurations occur with probabilities $p^3(1-p)p_s^2$, $p^3(1-p)p_s^2$ and $p^2(1-p)^2p_s^2$, respectively. In configuration (a), node j is mapped to $switch_{2j-2}$, thus leaving $proc_{1,2j-1}$ available (state s_1). In configuration (b), node j is mapped to $switch_{2j-1}$, thus leaving no processors available (state s_0). In configuration (c), node j is mapped to $switch_{2j}$, and one of $proc_{0,2i}$ or $proc_{1,2i}$ is needed to complete the embedding (state s_{-1}). Note that the transition probabilities do not depend on the particular index, j, of the node being mapped.

A second example is given in Fig. 15, where it is assumed that after iteration $j-1$, the process is in state s_{-1} (one of $proc_{0,2j}$ or $proc_{1,2j}$ is used). We show in Fig. 15(a), (b), and (c) three defect configurations for $T(j)$ that occur with probabilities $(1-p)^2$, $p^2(1-p)^2p_s$ and p^4p_s, respectively. The configuration of Fig. 15(a) leads to s_f since being in state s_{-1} requires that one of $proc_{0,2j}$ or $proc_{1,2j}$ is available. The configuration of Fig. 15(b) leads to s_{-2} while that of Fig. 15(c) leads to s_0.

Let σ_u^j be the probability of being in state s_u after the jth transition. Following the above mentioned enumeration procedure, we may find the state transition matrix A for the Markov process. To simplify the expression and derivation of A, we decompose it into:

$$A = A_{1,1}p_s^2 + A_{0,1}(1-p_s)p_s + A_{1,0}p_s(1-p_s) + A_{0,0}(1-p_s)^2,$$

$$\begin{bmatrix} \sigma_0^{j+1} \\ \sigma_1^{j+1} \\ \sigma_2^{j+1} \\ \sigma_{-1}^{j+1} \\ \sigma_{-2}^{j+1} \\ \sigma_f^{j+1} \end{bmatrix} = \begin{bmatrix} 4\pi_3 & 2\pi_3+\pi_2 & \pi_2+2\pi_1 & \pi_4 & 0 & 0 \\ \pi_4 & 2\pi_3 & 2\pi_3+4\pi 2 & 0 & 0 & 0 \\ 0 & \pi_4 & \pi_4+2\pi_3 & 0 & 0 & 0 \\ \pi_2 & \pi_2 & \pi_2 & 2\pi_3 & \pi_4 & 0 \\ 4\pi_2+2\pi_1 & 4\pi_2+2\pi_1 & 2\pi_1 & 2\pi_3+4\pi_2 & 2\pi_3 & 0 \\ \pi_2+2\pi_1+\pi_0 & 2\pi_1+\pi_0 & \pi_0 & 2\pi_2+4\pi_1+\pi_0 & 1-\pi_4-2\pi_3 & 1 \end{bmatrix} \begin{bmatrix} \sigma_0^j \\ \sigma_1^j \\ \sigma_2^j \\ \sigma_{-1}^j \\ \sigma_{-2}^j \\ \sigma_f^j \end{bmatrix}$$

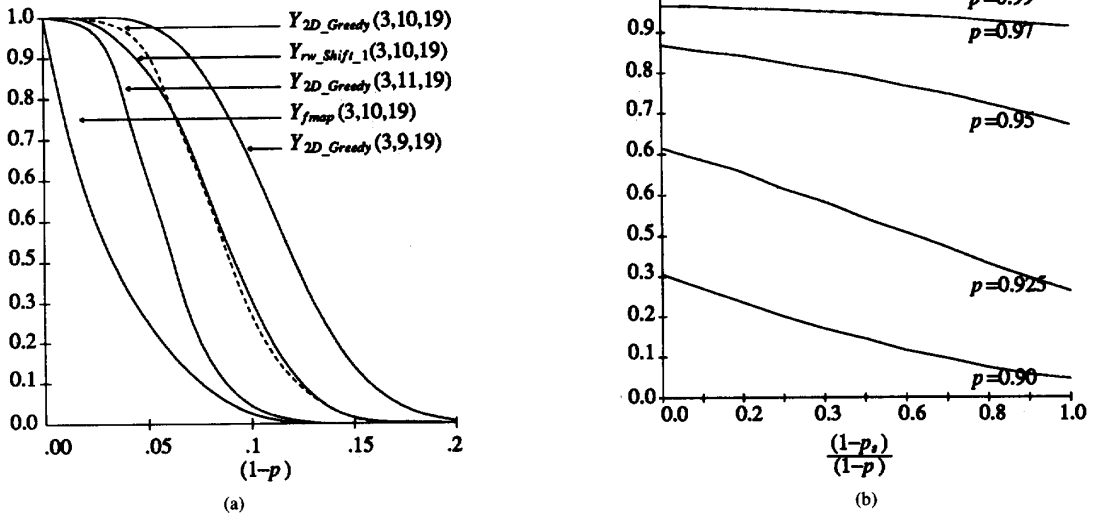


Fig. 16. The yields for row-wise Shift_1 and 2-D_Greedy. (a) The yield for different algorithms ($p_s = 0$). (b) The effect of faulty switches on $Y_{rw\_shift\_1}$.

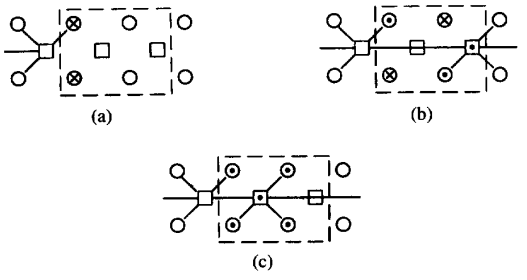


Fig. 15. States after iteration j assuming that, before iteration j, the state is s_{-1}. (a) s_f. (b) s_{-2}. (c) s_0.

where each of the four matrices corresponds to a particular pattern of failure of $switch_{2j-1}$ and $switch_{2j}$. The matrix $A_{1,1}$ corresponding to the two switches being nondefective is specified by the equation found at the bottom of the previous page where $\pi_4 = p^4$, $\pi_3 = p^3(1-p)$, $\pi_2 = p^2(1-p)^2$, $\pi_1 = p(1-p)^3$ and $\pi_0 = (1-p)^4$. The expressions for the other matrices are similar.

The Markov process is initiated at state s_0. This is because $proc_{0,-1}$ and $proc_{1,-1}$ are not available for embedding logical node 1 (they do not exist), and $proc_{0,0}$ and $proc_{1,0}$ are not used in the embedding of node 0 (node 0 does not exist). If n_s is odd, then the last node, n_l, cannot be shifted to the right. That is, the embedding is successful only if, after n_l iterations, the Markov process is in state s_0, s_1 or s_2. Hence, the probability of successful embedding is:

$$Y_{L\_Shift}(3, n_l, n_s) = (1\ 1\ 1\ 0\ 0\ 0) A^{n_l} (1\ 0\ 0\ 0\ 0\ 0)^T, \quad \text{if } n_s \text{ is odd.}$$

A slightly more complex expression may be obtained for even values of n. From this and (2), the success probability of the row-wise shift_1/$fmap^{1,1}$ may be computed. This probability is plotted in Fig. 16(a). In this Fig. the curve labeled $Y_{fmap}(3, 10, 19)$ is the probability that $fmap^{1,1}$ embeds successfully a 10 × 10 array onto a 19 × 19 defective PSVA without any reconfiguration. It is given by .

$$Y_{fmap}(3, 10, 19) = (p^4 p_s + 4p^3(1-p)p_s)^N$$

Where $N = n_l^2 = 100$. This curve is used to demonstrate the yield enhancement due to reconfiguration. In Fig. 16(b), we show the effect of switch faults on $Y_{rw\_Shift\_1}(3, 10, 19)$. As expected, the yield decreases for higher probability of switch faults.

In order to compare the 2-D_Greedy and the row-wise Shift_1 algorithms, we also plot $Y_{2-D\_Greedy}(3, 10, 19)$ in Fig. 16(a). This is the probability of successfully embedding a logical array of size at least 10 × 10 onto the 19 × 19 PSVA using 2-D_Greedy. Although the two yields are approximately equal, the yield of 2-D_Greedy is slightly lower for small values of p. The performance of 2-D_Greedy relative to Shift_1 improves as p increases. However, comparing 2-D_Greedy and Shift_1 for a fixed size logical array is misleading. In fact, even for the values of p for which $Y_{2-D\_Greedy}(3, 10, 19)$ is smaller than $Y_{rw\_Shift}(3, 10, 19)$, 2-D_Greedy has the advantage of being able, with some probability, to embed arrays larger than 10 × 10. For instance, as seen from Fig. 16(a), for $p = 0.95$, 2-D_Greedy may embed a 11 × 10 array in a 19 × 19 PSVA with probability 0.58. That is $Y_{2-D\_Greedy}(3, 11, 19) = 0.58$ while $Y_{rw\_Shift}(3, 11, 19) = 0$.

Next, we discuss two different variations of the Shift_1 embedding algorithm.

B. A More Flexible Shift_1/$fmap^{1,1}$ Embedding

In the previous section, we considered mapping functions in which each node may be shifted from its position in $fmap^{1,1}$

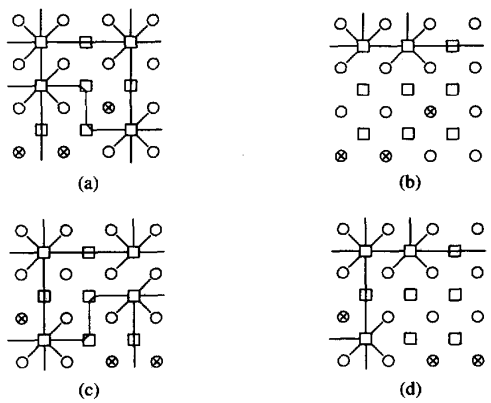


Fig. 17. Embedding 2 × 2 logical arrays onto defective 3 × 3 PSVA's (a), (c) $\Delta = (0,0)$ is tried first, (b), (d) $\Delta = (0,-1)$ is tried first.

only in the horizontal direction. With this restriction, however, the probability of successful embedding is maximized if, while embedding logical node (i,j), we try $\Delta(i,j) = (0,-1)$ then $(0,0)$ then $(0,1)$. In this section, we consider Shift_1/$fmap^{1,1}$ algorithms in which all five values of $\Delta(i,j)$ are tried. Using the same argument used for the row-wise case, it may be shown that trying the values of Δ in $\{(0,0),(-1,0),(0,-1)\}$ before those in $\{(0,1),(1,0)\}$ always maximizes the probability of successfully completing the embedding. However, there is no optimum order for trying the values of Δ from $\{(0,0),(-1,0),(0,-1)\}$. In some cases, compression, that is trying $\Delta = (-1,0)$ or $(0,-1)$ before $(0,0)$, may increase the probability of successfully completing the embedding. In some other cases, however, this may decrease that probability. We clarify this by the two examples in Fig. 17. For the PSVA of Fig. 17(a), the embedding is completed with $\Delta = (0,0)$ being tried before $(-1,0)$. If $\Delta = (-1,0)$ is tried first in this example, then, as shown in Fig. 17(b), node $(1,1)$ will be shifted to the left thus consuming $proc_{1,1}$ and causing the embedding of node $(2,1)$ to fail. A similar argument applies to Figures 17(c) and (d). In this case, however, the left shift of node $(1,1)$ prevents the upward shift of node $(2,2)$ because this shift would violate the conditions of Proposition 2.

With the possibility of shifting in any of the four normal directions, the predicate *valid* in Algorithm Shift_1/$fmap^{1,1}$ should return true only if the conditions of Proposition 2 are satisfied. Given the values of $\Delta(i-1,j)$ and $\Delta(i,j-1)$, the predicate *valid* is specified by

$$\text{valid}(i,j,\Delta(i,j)) = \begin{cases} false, & \text{if } \Delta(i,j) = (\pm 1,0) \text{ and } \Delta(i,j-1) = (0,1), \\ false, & \text{if } \Delta(i,j) = (0,\pm 1) \text{ and } \Delta(i-1,j) = (1,0), \\ false, & \text{if } \Delta(i,j) = (-1,0) \text{ and } \Delta(i-1,j) = (0,\pm 1), \\ false, & \text{if } \Delta(i,j) = (0,-1) \text{ and } \Delta(i,j-1) = (\pm 1,0), \\ true, & \text{otherwise.} \end{cases} \tag{3}$$

In other words, (i,j) may not be shifted vertically if $(i,j-1)$ is shifted to the right and may not be shifted horizontally if $(i-1,j)$ is shifted downward. Moreover, (i,j) may be shifted upward only if $(i-1,j)$ is not shifted horizontally and may be shifted leftward only if $(i,j-1)$ is not shifted vertically.

In order to analyze the probability of success of Shift_1/$fmap^{1,1}$, we observe that the probability of successfully embedding a logical node depends on the conditions resulting from embedding the previous nodes. As in the case of L_Shift_1, the conditions after embedding logical node (i,j) may be expressed in terms of the number of processors among $\{proc_{2i-1,2j-1}, proc_{2i-1,2j-2}, proc_{2i-2,2j-1}\}$ that are not marked after iteration i,j. Unlike the linear case, however, a distinction should be made between un-marked processors that may be used when mapping logical node $(i,j+1)$ and those that may be used when mapping logical node $(i+1,j)$. Two variables, $R_h(i,j)$ and $R_v(i,j)$ are used for that purpose.

- $R_h(i,j)$: is the number of processors among $proc_{2i-1,2j-1}$ and $proc_{2i-2,2j-1}$ that are available after the embedding of node (i,j) to be used when embedding node $(i,j+1)$. A negative value of $R_h(i,j)$ indicates that the embedding of node (i,j) required a shift to the right and the use of $|R_h(i,j)|$ processors from $\{proc_{2i-1,2j}, proc_{2i-2,2j}\}$.
- $R_v(i,j)$: is the number of processors among $proc_{2i-1,2j-1}$ and $proc_{2i-1,2j-2}$ that are available after the embedding of node (i,j) to be used when embedding node $(i+1,j)$. A negative value of $R_v(i,j)$ indicates that the embedding of node (i,j) required a downward shift and the use $|R_h(i,j)|$ processors from $proc_{2i,2j-1}$, $proc_{2i,2j-2}$.

The value of the tuple $\langle R_h(i,j), R_v(i,j)\rangle$ may be used to indicate the state of the embedding after iteration (i,j). Each of R_h and R_v may be equal to one of the five values -2, -1, 0, 1 and 2. Thus, there are 25 possible values for $\langle R_h, R_v\rangle$. The following proposition restricts the possible values of $\langle R_h, R_v\rangle$.

Proposition 4: The only possible case in which both $R_h(i,j)$ and $R_v(i,j)$ are non zero, is the case $R_h = R_v = 1$.

Proof: For $R_h(i,j)$ to be equal to 2, -1 or -2 we should have $\Delta(i,j) = (0,-1)$, $(0,+1)$ or $(0,+1)$, respectively. That is the mapping of (i,j) should be shifted horizontally, which, by Proposition 2 (or equation (3)), prevents node $(i+1,j)$ from being shifted upward. Thus, the embedding of node $(i+1,j)$ may not use any of $proc_{2i-1,2j-1}$ or $proc_{2i-1,2j-2}$. Also, if (i,j) is shifted horizontally, then the embedding of node (i,j) does not use any of $proc_{2i,2j-1}$ or $proc_{2i,2j-2}$. Hence, $R_v(i,j) = 0$. By a similar argument we may show that if $R_v(i,j) = 2$, -1 or -2, then $R_h(i,j) = 0$. □

The following proposition puts more restrictions on the possible values of $\langle R_h, R_v\rangle$ if, in algorithm *Shift_1*, an attempt is made to map each node (i,j) to its position in $fmap^{1,1}$ before attempting any shift.

Proposition 5: Assume that algorithm *Shift*$_1$/$fmap^{1,1}$ is applied to a PSVA with nondefective switches. If $\Delta = (0,0)$ is tried before $\Delta = (-1,0)$, then $R_v(i,j) \le 1$. Similarly, if $\Delta = (0,0)$ is tried before $\Delta = (0,-1)$, then $R_h(i,j) \le 1$. Moreover, in either of the above two cases, $R_v(i,j) = R_h(i,j) = 1$ only if $\Delta(i,j) = (0,0)$.

Proof: We start the proof by two observations.

O1) $R_v(i,j) = 2$ only if (i,j) is shifted upward ($\Delta(i,j) = (-1,0)$) and both $proc_{2i-1,2j-1}$ and $proc_{2i-1,2j-2}$ are nondefective.

O2) If node (i,j) is shifted horizontally, then $R_v(i,j) = 0$ because, in this case, according equations (3), node $(i+1,j)$ may not be shifted upward, and thus may not use any of $proc_{2i-1,2j-1}$ or $proc_{2i-1,2j-2}$.

We use these observations to prove, by induction on i, that if $\Delta = (0,0)$ is tried before $\Delta = (-1,0)$, then $R_v(i,j) \leq 1$ and $R_v(i,j) = 1$ only if $\Delta(i,j) = (0,0)$. The second part of the proposition may be proved in a similar way.

Clearly, node $(1,j)$ cannot be shifted upward, and thus $R_v(1,j) \neq 2$ and $R_v(1,j) = 1$ only if $\Delta(1,j) = (0,0)$. Next, if the induction hypothesis is true for $i-1$, then $R_v(i-1,j) \leq 1$ and thus, at most one of $proc_{2i-3,2j-1}$ and $proc_{2i-3,2j-2}$ may be used when embedding node (i,j). We consider three cases. The first case is when the embedding of node (i,j) is accomplished with $\Delta(i,j) = (0,\pm 1)$. In this case, the result follows directly from observation O2. The second case is when the embedding of node (i,j) is accomplished with $\Delta(i,j) = (0,0)$. In this case, by observation O1, $R_v(i,j) \neq 2$ and the result follows directly. Finally, the third case is when the embedding is accomplished with $\Delta(i,j) = (-1,0)$, which is true only if the embedding fails with $\Delta(i,j) = (0,0)$. In this case, noting that $R_v(i-1,j) \leq 1$, both $proc_{2i-2,2j-1}$ and $proc_{2i-2,2j-2}$ should be nondefective in order to have a total of three nondefective processors connected to $switch_{2i-2,2j-1}$. This, together with the fact that $R_h(i,j-1)$ is not negative (otherwise node $(i,j-1)$ is shifted to the right and (i,j) cannot be shifted upward) imply that both $proc_{2i-1,2j-1}$ and $proc_{2i-1,2j-2}$ are defective because, otherwise, there will be more than two available, nondefective, processors connected to $switch2i-1,2j-1$ and the embedding with $\Delta(i,j) = (0,0)$ would have been successful. Hence, if $\Delta(i,j) = (-1,0)$, then $R_v(i,j) = 0$, which proves the induction step. □

Corollary 1: If $\Delta = (0,0)$ is tried before $(-1,0)$ and $(0,-1)$ in algorithm Shift_1/$map^{1,1}$, then it is always possible to mark the processors such that if either $R_v(i,j)$ or $R_h(i,j)$ is equal to one, then the other is also equal to one.

Proof: From Proposition 5, if either $R_v(i,j)$ or $R_h(i,j)$ is equal to one, then $\Delta(i,j) = 0$, and the four processors connected to $switch_{2i-1,2j-1}$ are nondefective. According to the priorities in step 2.2 of Shift$_1$/$fmap^{1,1}$, $proc_{2i-1,2j-1}$ will not be marked and will be available to be used when either $(i,j+1)$ or $(i+1,j)$ are embedded. Thus $R_v(i,j) = R_h(i,j) = 1$. □

We will analyze two versions of the general Shift_1/$fmap^{1,1}$ embedding algorithm. In the first version, G1_Shift_1/$fmap^{1,1}$, the values of Δ are tried in the order $(0,0)$, $(0,-1)$, $(-1,0)$, $(0,1)$ and $(1,0)$. That is the no-shift position is tried first. In the second version, $G2$_Shift_1/$fmap^{1,1}$, the values of Δ are tried in the order $(0,-1)$, $(0,0)$, $(-1,0)$, $(0,1)$ and $(1,0)$. That is the left shift position is tried first. The analysis is simplified by assuming nondefective switches ($p_s = 1$). As clear from the analysis of L_Shift_1, the same type of analysis may be applied to the case $p_s \neq 1$.

When G1_Shift/$fmap^{1,1}$ is applied to PSVA's with nondefective switches, Propositions 4 and 5 and Corollary 1, imply that only six values are possible for the tuples $\langle R_h(i,j), R_v(i,j)\rangle$. Namely, $\langle 0,0\rangle$, $\langle 1,1\rangle$, $\langle 0,-1\rangle$, $\langle 0,-2\rangle$, $\langle -1,0\rangle$ and $\langle -2,0\rangle$. Using these tuples, we define the following states that include information about the consequences of embedding logical node (i,j):

$s_{0,1}(i,j)$: indicates that $\langle R_h(i,j), R_v(i,j)\rangle = \langle 0,0\rangle$,
$s_2(i,j)$: indicates that $\langle R_h(i,j), R_v(i,j)\rangle = \langle 1,1\rangle$ (possible only if $\Delta(i,j) = (0,0)$),
$s_3(i,j)$ and $s_4(i,j)$: indicate that $\langle R_h(i,j), R_v(i,j)\rangle = \langle -1,0\rangle$ and $\langle -2,0\rangle$, respectively, (possible only if $\Delta(i,j) = (0,1)$),
$s_5(i,j)$ and $s_6(i,j)$: indicate that $\langle R_h(i,j), R_v(i,j)\rangle = \langle 0,-1\rangle$ and $\langle 0,-2\rangle$, respectively, (possible only if $\Delta(i,j) = (1,0)$),
$s_7(i,j)$: indicates that the algorithm failed to embed logical node (i,j).

If we denote by $T(i,j)$ the set containing the four processors connected to $switch_{2i-1,2j-1}$, then the probability of successfully embedding logical node (i,j) will depend on: 1) the probabilities of defect distributions among the processors in $T(i,j)$, and 2) the state that resulted from embedding nodes $(i-1,j)$, $(i,j-1)$ and $(i-1,j+1)$. These two factors will also determine the state after the embedding of node (i,j). Note that the state after the embedding of node $(i-1,j+1)$ is needed because node $(i-1,j+1)$ is embedded in $Shift\_1$ after node $(i-1,j)$. Thus, if $R_h(i-1,j) = R_v(i-1,j) = 1$, then node (i,j) may be shifted upward and use the processor left by node $(i-1,j)$ only if node $(i-1,j+1)$ has not been shifted left and has already used that processor. However, if node $(i-1,j+1)$ is shifted to the left, then, $R_v(i-1,j+1) = 0$ and, from Proposition 5, $R_h(i-1,j+1) = 0$. Hence, state $s_{0,1}(i-1,j+1)$ is the only state that is possible with either $\Delta(i-1,j+1) \neq (0,-1)$ or $\Delta(i-1,j+1) = (0,-1)$. In order to record whether or not a left shift had taken place, we divide state $s_{0,1}(i,j)$ into two states, namely;

$s_0(i,j)$ indicates that $\langle R_h(i,j), R_v(i,j)\rangle = \langle 0,0\rangle$ and $\Delta(i,j) \neq (0,-1)$
$s_1(i,j)$ indicates that $\langle R_h(i,j), R_v(i,j)\rangle = \langle 0,0\rangle$ and $\Delta(i,j) = (0,-1)$.

In Fig. 18, we show a few examples of states after the embedding of node (i,j). The examples are given for specific defect configurations for the four processors in $T(i,j)$. In each of these examples, the current state, $s_u(i,j)$, depends also on the states resulting from embedding nodes $(i,j-1)$, $(i-1,j)$ and $(i-1,j+1)$. For instance, $s_0(i,j)$ in Fig. 18(a) is reached only if the state after embedding node $(i,j-1)$ is not $s_3(i,j-1)$ or $s_4(i,j-1)$ and the state after embedding node $(i-1,j)$ is not $s_5(i-1,j)$ or $s_6(i-1,j)$. With the shown default configuration for $T(i,j)$, $s_4(i,j-1)$ or $s_6(i-1,j)$ would have resulted in $s_f(i,j)$. Either $s_3(i,j-1)$ or $s_5(i-1,j)$ would have resulted in $s_3(i,j)$ or $s_5(i,j)$, respectively. Finally, both $s_3(i,j-1)$ and $s_5(i-1,j)$ would have resulted in either $s_4(i,j)$ or $s_6(i,j)$ depending on the order in which *right* or *down* shifts are tried.

In Fig. 18(b), $s_0(i,j)$ is reached assuming $s_u(i,j-1)$, $u \neq 3$ or 4, $s_2(i-1,j)$ and $s_u(i-1,j+1)$, $u \neq 1$. That is assuming that node $(i,j-1)$ is not shifted right, node $(i-1,j)$ has an extra processor and that extra processor has not been

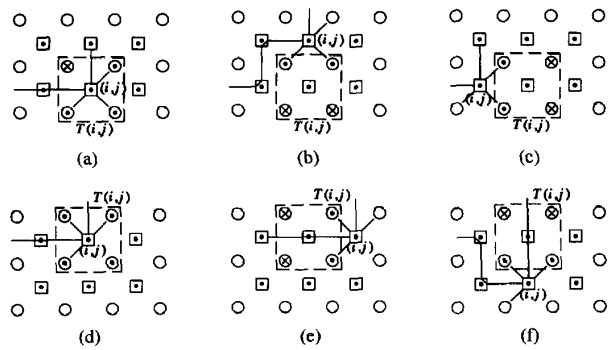


Fig. 18. Possible states after the embedding of node (i,j). (a) $s_0(i,j)$, (b) $s_1(i,j)$, (c) $s_2(i,j)$, (d) $s_0(i,j)$, (e) $s_3(i,j)$, (f) $s_5(i,j)$

used by node $(i-1,j+1)$. In Fig. 18(c), $s_1(i,j)$ is reached assuming $s_2(i,j-1)$ and $s_u(i-1,j)$, $u \neq 5$ or 6. The states $s_2(i,j)$, $s_3(i,j)$ and $s_5(i,j)$ shown in Fig. 18(d), (e), and (f), respectively, are reached assuming $s_u(i,j-1)$, $u \neq 3$ or 4 and $s_u(i-1,j)$, $u \neq 5$ or 6.

A Markov process similar to the one used to analyze L_Shift_1 may not be directly applied to compute the probability of successfully completing the embedding. This is because the result of iteration i,j not only depends on the previous iteration, $i,j-1$, but also on iterations $i-1,j$ and $i-1,j+1$. That is the process needs to remember the results of the previous n iterations. An alternative technique is to assume that $P_u(i,j)$ is the probability of being in state $s_u(i,j)$ after the embedding of node (i,j). It is then possible to write a set of recursive equations relating $P(i,j)$ with $P(i-1,j)$, $P(i-1,j+1)$ and $P(i,j-1)$, where $P(i,j)$ is a vector containing the eight probabilities $P_u(i,j)$ in the order $u = 0,\cdots,7$. By solving the resulting system of equations, we can calculate the probability that all the nodes are mapped successfully. The probability equations are derived by enumerating all possible configurations of defects in the elements of $T(i,j)$. For example, if π_u, $u = 1,\cdots,4$, are defined as in Section V-A, then, from our comment on Fig. 18(d), the probability of being in state $s_2(i,j)$ after the embedding of logical node (i,j) is given by:

$$P_2(i,j) = \pi_4 \overline{P}_{5,6}(i-1,j)\overline{P}_{3,4}(i,j-1)$$

where for simplicity of notation we used $P_{u,v} = P_u + P_v$, and $\overline{P}_{u,v} = (1 - P_{u,v})$. That is, the embedding of node (i,j) is completed with $\Delta(i,j) = (0,0)$ and $R_h(i,j) = R_v(i,j) = 1$ if and only if the four processors in $T(i,j)$ are nondefective (with probability π_4), node $(i-1,j)$ is not shifted downward (with probability $1 - P_{5,6}(i-1,j)$) and node $(i,j-1)$ is not shifted rightward (with probability $1-P_{3,4}(i,j-1)$). Similarly, the probabilities of being in the seven other states are:

$$\begin{aligned}
P_0(i,j) &= 4\pi_3 \overline{P}_{5,6}(i-1,j)\overline{P}_{3,4}(i,j-1) \\
&\quad + \pi_4 P_5(i-1,j)\overline{P}_{3,4}(i,j-1) \\
&\quad + \pi_4 \overline{P}_{5,6}(i-1,j)P_3(i,j-1) \\
&\quad + \pi_2 P_2(i-1,j)\overline{P}_{3,4}(i,j-1)\overline{P}_1(i-1,j+1) \\
P_1(i,j) &= \pi_2 \overline{P}_{5,6}(i-1,j)P_2(i,j-1))
\end{aligned}$$

$$\begin{aligned}
P_3(i,j) &= \pi_2 \overline{P}_{5,6}(i-1,j)\overline{P}_{3,4}(i,j-1) \\
&\quad + 2\pi_3(1 - P_{5,6}(i-1,j))P_3(i,j-1) \\
&\quad + \pi_4 \overline{P}_{5,6}(i-1,j)P_4(i,j-1) \\
P_4(i,j) &= 4\pi_2 \overline{P}_{5,6}(i-1,j)P_3(i,j-1) \\
&\quad + 2\pi_3 \overline{P}_{5,6}(i-1,j)P_4(i,j-1) \\
&\quad + 2(\pi_1 + \pi_2)\overline{P}_{5,6}(i-1,j)\overline{P}_{3,4}(i,j-1) \\
P_5(i,j) &= \pi_2 \overline{P}_{5,6}(i-1,j)\overline{P}_{3,4}(i,j-1) \\
&\quad + 2\pi_3 P_5(i-1,j)\overline{P}_{3,4}(i,j-1) \\
&\quad + \pi_4 P_6(i-1,j)\overline{P}_{3,4}(i,j-1) \\
P_6(i,j) &= 4\pi_2 P_5(i-1,j)\overline{P}_{3,4}(i,j-1) \\
&\quad + 2\pi_3 P_6(i-1,j)\overline{P}_{3,4}(i,j-1) \\
&\quad + \pi_1 \overline{P}_{5,6}(i-1,j)\overline{P}_{3,4}(i,j-1) \\
P_7(i,j) &= 1 - P_0(i,j) - P_1(i,j) - P_2(i,j) \\
&\quad - P_{3,4}(i,j) - P_{5,6}(i,j).
\end{aligned}$$

Now, by setting $P(0,j) = P(i,0) = (10000000)^T$ for any i and j, we may iteratively compute $P(i,j)$, $i,j = 1,\cdots,n_l$. Assuming that n_s is odd, then no node at the right boundary should be shifted rightward, and no node at the bottom boundary should be shifted downward. That is,

$$\begin{aligned}
&Y_{G1_shift}(3,n_l,n_s) \\
&\quad = [\Pi_{i=1}^{nl-1}\Pi_{j=1}^{nl-1}\overline{P}_7(i,j)] \\
&\quad\quad * [\Pi_{j=1}^{nl-1}\overline{P}_{5,6,7}(n_l,j)] \\
&\quad\quad * [\Pi_{i=1}^{nl-1}\overline{P}_{3,4,7}(i,n_l)] * P_{0,1,2}(n_l,n_l).
\end{aligned}$$

A slightly more complex expression may be derived for the case of even n_s.

The probability $Y_{G1\_Shift}(3,10,19)$ is plotted in Fig. 19 for the embedding of an 10×10 logical array onto a 19×19-switch PSVA. By comparing the results for G1_Shift and row-wise Shift_1, we note that the yield of the latter is greater than the yield of the former. Given that G1_Shift allows shifting in all four directions while $rw - Shift$ allows only horizontal shifts, it is clear that the compression toward the left (trying $\Delta = (0,-1)$ first) is the reason for the superiority of rw_Shift. It is possible to combine the advantages of both algorithms, that is to allow shifting in all directions and at the same time compress toward the left. This is achieved by trying the values of $\Delta(i,j)$ in algorithm Shift_1/$fmap^{1,1}$ in the order $(0,-1)$, $(0,0)$, $(-1,0)$, $(0,1)$ and $(1,0)$. This algorithm is called G2_Shift_1/$fmap^{1,1}$.

From Propositions 4 and 5, the permissible values of $\langle R_h(i,j), R_v(i,j)\rangle$ after embedding node (i,j) using G2_Shift_1 are: $\langle 0,0\rangle$, $\langle 1,0\rangle$, $\langle 2,0\rangle$, $\langle 1,1\rangle$, $\langle 0,-1\rangle$, $\langle 0,-2\rangle$, $\langle -1,0\rangle$ and $\langle -2,0\rangle$. As in G1_Shift_1, two states need to be defined for $\langle R_h, R_v\rangle = \langle 0,0\rangle$. Adding a failure state, the total number of states needed to analyze G2_Shift_1/$fmap^{1,1}$ adds up to 10 states. We have derived the equations relating the probabilities for these states and have computed the yield, $Y_{G2\_shift\_1}(3,n_l,n_s)$. The results for the embedding of a 10×10 logical array onto a 19×19 PSVA is shown in Fig. 19. As expected, G2_Shift_1 outperforms row-wise Shift_1. The yield improvement is, however, very modest.

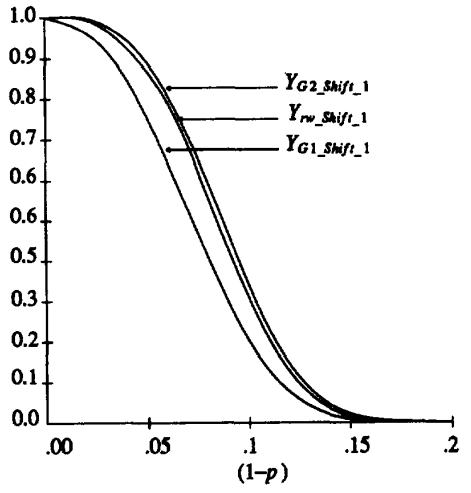


Fig. 19. The yield for embedding 10 × 10 array onto a 19 × 19 defective PSVA.

VI. Run Time Reconfiguration of PSVA

Run-time faults may be dealt with in PSVA's by providing additional redundancy at each node. Specifically, if d redundancy is desired for proper operation ($d = 2$ for fault detection and 3 for fault masking), then each logical node is mapped to the PSVA with ρ redundancy for $\rho > d$. With this, $\rho - d$ processors per logical node are available to be used as spares. For example, if triple redundancy is desired, then the mapping defined by (1) may be used to map a logical array onto a nondefective PSVA such that four processors constitute each logical node; three processors to be used in a TMR mode and the fourth to be used as a spare to replace any of the other three. This idea of using sparing along with TMR has been used in the design of FTMP [7], where a pool of spares is made available to replace any faulty unit in a triply redundant module. In PSVA's, however, the use of spares is restricted in the sense that a spare can only replace one of three specific processors.

In order to be more specific, define a *d-deficient* logical node to be a node with more than $\rho - d$ faults in the processors constituting it. Such a node has less than d nonfaulty processors left and thus cannot operate properly in a d-redundant mode. Because faults may not occur uniformly at run-time, some node may become *d-deficient* while some other node may have less than $\rho - d$ faults in the processors constituting it. In other words, the system may fail not because of lack of spares, but because the hardware interconnections restrict the set of processors that may be replaced by each spare. This restriction greatly reduces the survivability of the system.

In order to improve the survivability, a run-time reconfiguration algorithm may be invoked when a node becomes *d-deficient*. The goal of the reconfiguration is to restore the required redundancy at each node. Ideally, a run-time reconfiguration algorithm should be a simple algorithm which may be executed distributively and which only requires the remapping of *deficient* nodes (constant time complexity). A good candidate for such reconfiguration is a run-time adaptation of the Shift_1 algorithm of Section V; when a logical node, (i, j), which is mapped to $switch_{map}(i, j)$, becomes *deficient*, then remap this node to one of the four switches $switch_{\text{map}(i,j)+\Delta}$, where $\Delta = (0, \pm 1)$ or $(\pm 1, 0)$.

A shift of node (i, j) in one of the four directions is successful only if the neighboring logical node along that direction, say (i', j'), has an extra spare that it may give up and that may be used by (i, j). If this is not the case, however, then it may still be possible to shift (i, j) if (i', j') is successfully shifted along the same direction, thus relinquishing one or more processors to (i, j). This process may be continued recursively until we either can shift a node successfully, or we reach a node which may not be shifted. For example, assume that an $n_l \times n_l$ logical array is initially embedded in an $n_s \times n_s$ PSVA using $fmap^{1,1}$ and that triple redundancy is required. Hence, when a node (i, j) becomes 3-*deficient* the following algorithm may be used to try to shift node (i, j) by one switch to the left:

Algorithm left_shift (i, j)
IF both $proc_{2i-2,2j-2}$ and $proc_{2i-1,2j-2}$ are faulty
OR $j = 1$ THEN return *not-successful*
  ELSEIF the four processors constituting node
  $(i, j-1)$ are not faulty, THEN
    shift (i, j) to $switch_{2i-1,2j-2}$ and return a set
    which contains the nonfaulty processors
    among $proc_{2i-2,2j-1}$ and $proc_{2i-1,2j-1}$.
    These are the processors relinquished when (i, j)
    is shifted to the left.
  ELSE call left_shift$(i, j-1)$;
      IF the call is successful and the set of
      processors relinquished due to left_shift$(i, j-1)$
      allows (i, j) to be embedded in $switch_{2i-1,2j-2}$,
      then return a set which contains the nonfaulty
      processors among $proc_{2i-2,2j-1}$ and
      $proc_{2i-1,2j-1}$.
      ELSE return *not-successful*.

If left_shift is not successful, then a similar up-shift is tried, then a right_shift and finally a down_shift. The resulting reconfiguration algorithm is called $(n_l - 1)$-compress because the recursive shifting of nondeficient nodes along some direction is equivalent to compressing the embedding in that direction. At run time, this compression may require the remapping of up to $n_l - 1$ nodes.

The above reconfiguration is triggered by a node that becomes 3-*deficient*. Such a node is capable of participating in the reconfiguration since it still has two nonfaulty processors. These processors can detect the *deficiency* status by periodically examining the status word of the switch to which they are actively connected. The other nodes that are involved in the reconfiguration are invoked by message passing and execute in a distributive manner. Processors may determine the active connections of neighboring processors by examining the status word of a shared switch. They may also change such connections by writing onto that status word.

The run-time complexity of $(n_l - 1)$-compress is $O(n_l)$. This may be reduced if the nesting level of the recursive calls to left_shift, up_shift, right_shift and down_shift is limited to m levels, for some $m < n_l$. The resulting algorithm is

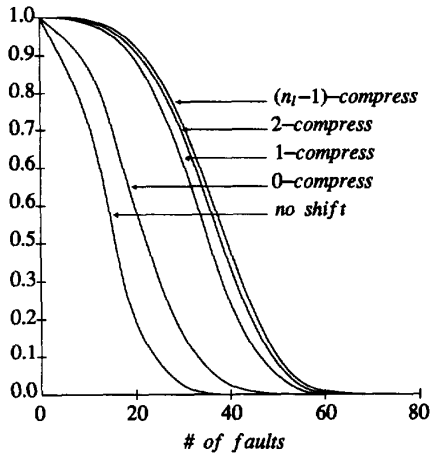


Fig. 20. Probability of surviving run time fault in a 19 × 19 PSVA.

called m-compress. Clearly, 0-compress attempts only to shift the deficient node and thus is fast. Its probability of success, however, is expected to be low compared to that of $(n_l - 1)$-compress.

In order to study the effect of m on the success of the reconfiguration algorithm, we simulated the application of m-compress for the reconfiguration of an 10×10 logical array which is embedded in an 19×19 PSVA using $fmap^{1,1}$ with $d = 3$. Clearly, only 300 of the 400 processors in the PSVA are used in the embedding and the remaining 100 processors may be used as spares. In Fig. 20, we show the probability that m-compress successfully reconfigures the array to tolerate k randomly chosen faulty processors. As seen from the figure, compression with low values of m gives good results. In fact, for $m = 2$, any increase in the value of m results in a minute improvement in system survivability.

VII. Conclusion

We have presented a versatile architecture for implementing multiple redundancy in computational arrays, and we have studied different embedding and reconfiguration algorithms that may be applied to this architecture. All the algorithms may be applied either to tolerate initial defects or to recover from run time faults. In the paper, however, we distinguished between defect tolerance algorithms and run time algorithms according to the amount of global restructuring that is required. This distinction is natural because the algorithms that require global restructuring outperform those that are restricted to local restructuring, but are less suitable for run time reconfiguration.

Given the flexibility of the PSVA architecture, we emphasized the strategies underlying the reconfiguration algorithms and the techniques used to analyze them. These strategies and techniques may be applied to different instances of the restructuring problem. For example, we analyzed the embedding and reconfiguration algorithms assuming that triple redundancy is required for proper operation. Also, the Shift_1 algorithms were analyzed assuming that $fmap^{1,1}$ is the initial fixed mapping. The same type of analysis may be applied if different degrees of redundancy are required and different initial mappings are used.

Although it is assumed that switch/voting elements may fail, those elements are not used in a redundant mode, and thus should be self-checking elements. This also implies that errors in switch/voters cannot be masked. Multiple redundancy schemes for processor arrays may be realized without the use of switch/voter elements. For this, direct connections should be provided among the d processors constituting a logical node to allow the results of each processor to be transmitted to the other $d - 1$ processors. The voting is thus performed independently and distributively by each processor [10]. Among the previously proposed processor arrays, two architectures may be used to realize such a distributed multiple redundancy approach. Namely, the iWarp [17] and the CHIP [22]. In the former, direct connections may be realized by time multiplexing individual links and in the latter, direct connections may be established by reconfiguring a mesh of communication tracks. Mapping a logical node into d processors in the physical array will no longer be constrained by the requirement that the d processors be physically adjacent, but rather by the allowable degree of multiplexing (in iWarp) and by the flexibility/track-width of the interconnection system (in CHIP-like architectures). Although studying and analyzing multiple redundancy mappings in iWarp and CHIP are beyond the scope of this paper, the ideas, algorithms and analysis techniques used for multiple redundancy mappings in PSVA's are useful starting points for such studies.

The study of multiple redundancy in PSVA's is the first step toward studying the applicability of processor meshes as general purpose, fault tolerant, multiprocessors. These systems are appealing because of the ability to selectively set the degree of redundancy according to the required reliability and the possibility of dynamically reconfiguring the system after faults to efficiently utilize the available redundancy.

Appendix
Glossary of Notation

Section II

$switch_{i,j}$	the switch located in row i, column j of a PSVA, starting with row and column 1,1.
$proc_{i,j}$	the processor located in row i, column j of a PSVA, starting with row and column 0,0.
$C_{i,j}$	the active set —the set of processors with active connections to $switch_{i,j}$.
$c_{i,j}$	the cardinality of active set $C_{i,j}$.
Maj	the single bit majority function used within the switch/voters of a PSVA.
$map(i,j)$	a function which specifies a mapping from a logical node (i,j) to a physical PSVA switch. The function maps from a row-column pair in the logical array to a row-column pair in the physical PSVA.
$map_x(i,j)$	the physical row that logical node (i,j) is mapped into.
$map_y(i,j)$	the physical column that logical node (i,j) is mapped into.

p the probability that a processor is nonfaulty.
p_s the probability that a switch is nonfaulty.

Section III

Shift_1 a mapping from logical nodes to physical switches which allows single switch position deviance from a given fixed mapping.
$fmap(i,j)$ the fixed mapping from logical nodes to physical switches utilized in the Shift_1 mapping.
$\Delta(i,j)$ the offset from the fixed mapping utilized in Shift_1. The values for $\Delta(i,j)$ may be (0,0) for no shift, (1,0) for a shift down, (0,1) for a shift right, (-1,0) for a shift up, or (0,-1) for a shift left.
d_x the difference of the row values in the mappings of nodes (i,j) and (i-1,j).
d_y the difference of the column values in the mappings of nodes (i,j) and (i-1,j).
$\overline{d}_x$ the difference of the row values in the mappings of nodes (i,j) and $(i,j-1)$.
$\overline{d}_y$ the difference of the column values in the mappings of nodes (i,j) and $(i,j-1)$.

Section IV

OK(x) a boolean function which returns true if component x is nonfaulty.
H_k the set of unmarked (or available) processors connected to $switch_k$ in a linear PSVA. The set is indicated by two subscripts (for row and column) in two-dimensional PSVA's.
$R(j)$ the number of nondefective processors remaining on the right side of $switch_{map(j)}$ after logical node j has been mapped onto it. These processors may be used in the mapping of logical node $j+1$ if necessary. $R(j)$ may have the values 0, 1, or 2.
map_{opt} a mapping of logical nodes onto a linear PSVA which maximizes the number of mapped logical nodes.
$Y_S(d,n_a,n_s)$ the yield obtained when algorithm S is used to map n_a logical nodes into a linear PSVA of n_s switches maintaining redundancy d per node.
$s_{y,u}$ a state in the layered Markov chain used to analyze the Greedy algorithm. Subscript y indicates the number of nodes successfully mapped, and subscript u indicates the number of processors available for mapping the next node.
$\sigma^k_{y,u}$ the probability that the Greedy mapping is in state $s_{y,u}$ after k switches have been examined (after the kth transition).
σ^k a vector of all the probabilities $\sigma^k_{y,u}$. Subscript y ranges from 0 to n_a and subscript u ranges from 0 to 2. Thus, the vector contains the $3(n_a+1)$ different probabilities of being in the various mapping states after the kth transition.

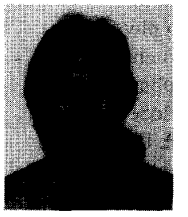
Section V

$fmap^{u,v}$ a mapping of logical nodes into a PSVA such that u consecutive switches are used, followed by v consecutive switches being unused, and repeating the pattern for each row.
$\mu(fmap^{u,v},d)$ the *utilization factor* of mapping $fmap^{u,v}$ with redundancy d. This value indicates the percentage of processors in the PSVA which are used by the mapping.
$valid(i,j,\Delta(i,j))$ a boolean function which returns true if shifting the mapping of logical node (i,j) by $\Delta(i,j)$ still allows correct connections between the node and its west and north neighbors.
$s_f, s_{-2}, \cdots, s_2$ the states used in the Markov chain used to analyze the row-wise Shift_1 algorithm. s_f is the failure state. s_{-2} and s_{-1} indicate that 2 and 1 processors, respectively, must be borrowed. s_0 indicates no borrowing is needed. s_1 and s_2 indicate that 1 and 2 processors, respectively, are available.
π_i the probability that exactly i processors out of four are nonfaulty on a switch.
$R_h(i,j)$ the number of processors on $switch_{map(i,j)}$ available, if necessary, for the mapping of logical node $(i,j+1)$ after the mapping of logical node (i,j). A negative value (-1 or -2) indicates that node (i,j) was shifted to the right and borrowed 1 or 2 processors from its neighboring switch.
$R_v(i,j)$ analogous to $R_h(i,j)$ but for vertical neighbors.
$s_0(i,j),\cdots,s_7(i,j)$ states based upon the different combinations of values attainable by R_h and R_v in the G1_Shift_1 algorithm after the mapping of logical node (i,j).
$P_0(i,j),\cdots,P_7(i,j)$ probabilities corresponding to the states $s_0(i,j),\cdots,s_7(i,j)$.

REFERENCES

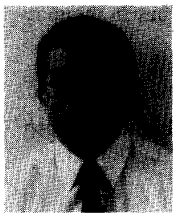
[1] "Paragon XP/S product overview," *Intel Corporation*, 1991.
[2] J. Abraham, P. Banerjee, C. Chen, W. Fuchs, S.Y. Kuo, and A. Reddy, "Fault tolerance techniques for systolic arrays," *IEEE Comput.*, vol. 20, no. 7, pp. 65–74, July 1987.
[3] M. Alam and R. Melhem, "An efficient spare allocation scheme and its application to fault tolerant binary hypercubes," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 2, no. 1, pp. 117–126, Jan. 1991.
[4] C. Anfinson and F. Luk, "A linear algebraic model of algorithm-based fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1599–1604, Dec. 1988.
[5] M. Chean and J. Fortes, "The full-use-of-suitable-spares (FUSS) approach to hardware reconfiguration for fault-tolerant processor arrays," *IEEE Trans. Comput.*, vol. 39, no. 4, Apr. 1990.
[6] R. Harper, J. Lala and J. Deyst, "Fault tolerant parallel processor architecture overview," in *Proc. of FTCS 18*, 1988, pp. 252–257.
[7] A. Hopkins, "FTMP—A Highly Reliable Fault Tolerant Multiprcessor for Aircraft," *Proc. of IEEE*, vol. 66, no. 10, pp. 1221–1239 1978.

[8] K. Huang and J. Abraham, "Algorithm-based fault-tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-36, no. 6, pp. 518–528, June 1984.
[9] J. Kim and S. Reddy, "On the design of fault tolerant tow dimensional systolic arrays for yield enhancement," *IEEE Trans. Comput.*, vol. 38, no. 4, pp. 515–525, Apr. 1989.
[10] D. Kiskis and K. Shin, "Embedding triple-modular redundancy into a hypercube architecture," in *Proc. of the Third Conf. on Hypercube Concurrent Comput. and Applicat.*, 1988, pp. 337–245.
[11] I. Koren, "A reconfigurable and fault-tolerant VLSI multiprocessor array," in *Proc. 8th Symp. Comput. Architecture*, 1981, pp. 425–442.
[12] S. Y. Kung, S. N. Jean, and C.W. Chang, "Fault-tolerant array processors using single track switches," *IEEE Trans. Comput.*, vol. 38, no. 4, pp. 501–514, Apr. 1989.
[13] C. Kwan and S. Toida, "Optimal fault-tolerant realizations of some classes of hierarchical tree systems," in *Proc. of FTCS 11*, 1981, pp. 176–178.
[14] E. Manolakos, D. Dakhil, and M. Vai, "Concurrent error diagnosis in mesh array architectures based on overlapping H-processes," in *Proc. Int. Workshop on Defect and Fault Tolerance on VLSI Syst.*, 1991, pp. 139–152.
[15] R. Melhem and J. Ramirez, "Meshes with multiple redundancy," in *Algorithms and Parallel VLSI Architectures II*, P. Quinton and Y. Robert, Eds. New York: Elsevier, 1991.
[16] R. Melhem, "Bi-level reconfigurations of fault tolerant arrays," *IEEE Trans. Comput.*, vol. 41, no. 2, pp. 231–239, Feb. 1992.
[17] O. Menzilcioglu, H. T. Kung, and S. Wong, "Comprehensive evaluation of a two-dimensional configurable array," in *Proc. Fault Tolerant Computing Symp.*, 1989, pp. 93–100.
[18] R. Negrini, M. Sami, and R. Stefanelli, "Fault tolerance techniques for array structures used in supercomputing," *IEEE Comput.*, vol. 19. no. 2, pp. 78–87, Feb. 1986.
[19] A. Rosenberg, "The Diogenes approach to testable fault-tolerant array processors," *IEEE Trans. Comput.*, vol. C-32, no. 10, pp. 902–910, Oct. 1983.
[20] L. Shombert and D. Siewiorek, "Using redundancy for concurrent testing and repairing systolic arrays," in *Seventeenth Int. Symp. on Fault-Tolerant Computing*, 1987, pp. 244–249.
[21] A. Singh and H. Youn, "An efficient restructuring approach for wafer scale processor arrays," in *Proc. Int. Workshop on Defect and Fault in VLSI Syst.*, 1988, pp. 395–407.
[22] L. Snyder, "Introduction to the configurable, highly parallel computer," *Comput.*, vol. 15, no. 1, pp. 47–56, Jan. 1982.
[23] N. Theuretzbacher, "VOTRICS: Voting triple modular computer systems," in *Proc. of FTCS 16*, 1986, pp. 144–150.



John C. Ramirez was born in Pittsburgh, PA on July 8, 1964. He is currently a Lecturer in Computer Science at the University of Pittsburgh and is also working to complete his Ph.D degree in computer science at Pitt. He received the B.S. degree in biochemistry and mathematics from Duquesne University in 1986 and the M.S. degree in computer science from the University of Pittsburgh in 1989.

In 1991, he was the recipient of the *Orrin E. and Margaret M. Taulbee Award for Excellence in Computer Science*. His research interests include fault-tolerant computing and parallel and distributed algorithms and architectures.



Rami G. Melhem (S'82–M'84) was born in Cairo, Egypt, in 1954. He received a B.E. in electrical engineering from Cairo University, Egypt, in 1976, an M.A. degree in mathematics and an M.S. degree in computer science from the University of Pittsburgh in 1981, and a Ph.D. degree in computer science from the University of Pittsburgh in December 1983.

Since 1989, he has been an Associate Professor of Computer Science at the University of Pittsburgh. Previously, he was an Assistant Professor at Purdue University and at the University of Pittsburgh. He has published numerous papers in the areas of systolic architectures, parallel computing, fault tolerant processor arrays and optical computing.

He served in program committees for several conferences and he is on the editorial board of the IEEE TRANSACTIONS ON COMPUTERS. He also edited a special issue of the *Journal of Parallel and Distributed Computing* on "Optical Computing and Interconnection Systems." Dr. Melhem is a member of the IEEE Computer Society, the Association for Computing Machinery, and the International Society for Optical Engineering.