A Polynomial Time Algorithm for Reconfiguring Multiple-Track Models

Theodora A. Varvarigou, Vwani P. Roychowdhury, and Thomas Kailath, *Fellow, IEEE*

***Abstract*— In this paper we address the issue of developing efficient algorithms for reconfiguring mesh-connected arrays of identical Processing Elements (PE's) in the presence of faulty PE's. In particular, we study a recently introduced multiple-track model that uses $m\frac{1}{2}$-track wide channels and m spare rows (or columns) along each boundary of the array. In order to distinguish this model from other multiple-track models that use less number of spare processors, we refer to it as a $m\frac{1}{2}$-track-m-spare model. For the special case of $m = 1$, polynomial time algorithms for reconfiguring the arrays have been recently developed. An exponential time algorithm for reconfiguring the special case of $2\frac{1}{2}$-track-2-spare model has also been developed. However, the problem of designing efficient algorithms for $m = 2$ and for arbitrary $m > 2$ was left open. In this paper we address this open problem and present a polynomial time algorithm for solving the combinatorial problem that underlies the reconfiguration procedure for arbitrary m.**

***Index Terms*— Array processors, compensation paths, fault tolerance, multiple tracks, reconfiguration.**

I. Introduction

In this paper we address the issue of developing efficient algorithms for reconfiguring mesh-connected arrays of identical Processing Elements (PE's) in the presence of faulty PE's. The particular model that we consider was first introduced in [5] and [4] (for other types of models and for efficient algorithms in such models see [7], [3], [10], and [9]); it consists of an $N \times L$ array of nonspare PE's, m rows (or columns) of spare PE's along each boundary, m tracks along every grid line (or routing channels), and multiple-track switches located at the intersections where processors are connected to the grid (see Fig. 1). It is further assumed that a faulty PE can be converted into a connecting element, thereby making an implicit assumption that there is an extra channel within every PE. The latter assumption has led to the nomenclature of an $m\frac{1}{2}$-track model; however, to emphasize the fact that such a model has m rows (or columns) of spare PE's along every boundary [10], we shall refer to it as a $m\frac{1}{2}$-track-m-spare model (this is to facilitate distinctions with other models that use only one spare row (or column) along each boundary).

Manuscript received June 10, 1991; revised August 28, 1992. This work was supported in part by the SDIO/IST U.S. Army Research Office under Contract DAAL03-87-K-0033.

T. A. Varvarigou is AT&T Bell Laboratories, Holmdel, NJ 07733.

V. P. Roychowdhury is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

T. Kailath is with the Department of Electrical Engineering, Information Systems Laboratory, Stanford University, Stanford, CA 94305.

IEEE Log Number 9207176.

Given an $N \times L$ array with faulty PE's, the issue is to reconfigure it so that a fault free $N \times L$ logical array can be constructed using the healthy nonspare PE's and some spare PE's (to replace the faulty ones). Moreover, the hardware needed to route the necessary connections should not exceed the available resources as described above. If such a reconfiguration is possible for the given array, we shall refer to it as *reconfigurable*.

A set of sufficient conditions for determining whether an array with a particular distribution of faulty processors is reconfigurable in the m$\frac{1}{2}$-track-m-spare model was derived in [5] and [4]. The sufficient conditions can be simply stated in terms of the so-called *compensation paths*. Let a nonspare PE at location (x, y) be faulty, then in any valid reconfiguration it has to be replaced by a healthy processor. Let the faulty PE at (x, y) be replaced by a healthy PE, say at location (x', y'), which in turn is replaced by a healthy PE, say at location (x'', y''); one can continue this chain until one ends up at a spare PE. Now a compensation path can be defined as the ordered sequence of nodes (x, y), (x', y'), (x'', y''), $\cdots$, involved in the replacement chain. A compensation path is called continuous if it runs along grid lines and does not skip any node. Based on this concept of a compensation path, the following set of sufficient conditions can be stated [5], [4], [6]:

An $N \times L$ array of nonspare PE's is reconfigurable into an $N \times L$ array of healthy PEs in a $m\frac{1}{2}$-track-m-spare model if there exists a set of continuous and straight compensation paths covering all the faulty nonspare PE's such that 1) at most m of the compensation paths overlap along any row or column, 2) compensation paths in different rows and columns do not intersect, and 3) there is no near-miss situation.

A near-miss situation between two adjacent rows (or columns) arises when l compensation paths that run in the same direction in one row (or column) of the array overlap with k compensation paths running in the opposite direction in the other row (or column), and $k + l > m$. One can easily show that if a near-miss situation arises then m tracks are not sufficient to handle the reconfiguration; hence near-miss situations are to be avoided.

Let F denote the set of faulty PE's, and $|F|$ the total number of such PE's. Then one can easily verify that checking whether a given array with $|F|$ faulty PE's can be reconfigured according to the sufficient conditions is equivalent to solving the following combinatorial problem:

Problem 1: Let V be the set of grid points in an $N \times L$ two-dimensional rectangular grid, and let $F \subset V$. Determine

0018-9340/93$03.00 © 1993 IEEE

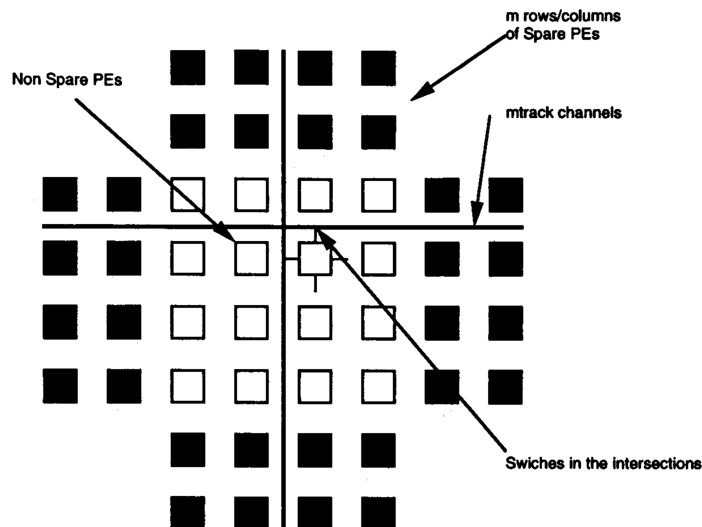


Fig. 1. The $m\frac{1}{2}$-track-m-spare reconfiguration model.

a set of straight lines such that

1) Each vertex $v \in F$ is assigned a *straight line* connecting it to one of the four boundaries of the grid.
2) At most m of these straight lines in the same row or column are overlapping.
3) The straight lines are nonintersecting, and there are no near-miss situations. Recall that a near-miss situation occurs if k lines in the same direction in one row (or column) overlap with l lines in the opposite direction in an adjacent row (or column) such that $k + l > m$.

For the special case of $m = 1$ (i.e., the $1\frac{1}{2}$-track-1-spare model), an exponential time algorithm was first developed in [6] by reformulating Problem 1 as a maximum independent set problem. However, the maximum independent set problem is NP-complete and the best known algorithms are of exponential time complexity. The problem of determining an efficient algorithm has recently been solved, and a polynomial time algorithm [the time complexity is $O(|F|^2)$] has been presented in [8]. Moreover, linear time algorithms for determining valid reconfigurations have been developed for the restricted cases where the spare processors are not present along all four sides of the array. More recently, an improvement of the above mentioned algorithm has resulted in an $O(|F| \log |F|)$ time-complexity algorithm [1].

For $m = 1$, the results mentioned above focus on the *satisfiability* question, i.e., whether *all the faulty PE's* can be covered by a set of straight and nonintersecting compensation paths. Often, however, a more relevant issue might be to determine the maximum number of faulty PE's for which such a covering set of compensation paths exists. In [2] a polynomial time algorithm of time complexity $O(|F|^3)$ has been developed for solving the corresponding combinatorial problem.

For the special case of $m = 2$ of Problem 1, an exponential time algorithm was presented in [5] and [4]; the algorithm is again based on reducing the corresponding combinatorial problem to a maximum-independent set problem. *However, no algorithm has been presented for the general case of arbitrary m.*

In this paper we address this open problem and develop a polynomial time algorithm for solving Problem 1 when m is arbitrary. The algorithm developed here is a generalization of the algorithm presented in [8] and has time complexity of $O(m|F|^2)$.

II. Efficient Algorithms for Reconfiguring $m\frac{1}{2}$-Track-m-Spare Models

In this section we shall present an efficient algorithm for solving Problem 1 that underlies the reconfiguration procedure in $m\frac{1}{2}$-track-m-spare models.

We can make the following observations about Problem 1:

1) If there are more than one consecutive rows (columns) that contain none of the vertices in F, then it is clear from the definition of the problem that all but one of these blank rows (columns) play no role in searching for a valid assignment. Hence, without loss of generality, we can delete such rows (columns) from the description of our problem and assume that $1 \leq N, L \leq 2|F|$.
2) Each vertex $v \in F$ can be assigned to at most one of four possible line segments, where each segment is along one of the four grid lines intersecting at v. Hence, instead of talking in terms of assigning line segments we can talk in terms of assigning directions, e.g., assigning a segment that connects a vertex v to the left side of the grid, can be interpreted as assigning the direction *Left* to the node v. In the rest of this paper we shall interchangeably use the two equivalent descriptions.

An *assignment* for Problem 1 is a mapping of every node in the set F to the set of four possible directions $D = \{Left, Right, Up, Down\}$. An assignment is a *valid assignment* if the corresponding line segments do not intersect, there are no more than m of them overlapping in the same row or column,

and there are no near-miss situations. Moreover, a direction d is said to be a *valid direction* for a node $v \in F$ if there is a valid assignment in which v is mapped to d.

Geometrically, the major differences between the case where m is arbitrary and where $m = 1$ (i.e., the $1\frac{1}{2}$-track-1-spare model [6], [8]) are as follows: 1) For the $m\frac{1}{2}$-track-m-spare case, up to m-lines can overlap along the same row or column, whereas in the single-track case only a single line is allowed along every column or row. 2) The near-miss situation for the $m\frac{1}{2}$-track-m-spare case is much more complicated.

The methodology used in [8] for solving Problem 1 for $m = 1$, comprised two steps: 1) Developing linear time algorithms for several special cases where the permitted directions are restricted. 2) Using the algorithms for the special cases to solve the general case. In this section, we shall use the same approach and show how the algorithms for the various cases can be modified to accommodate the above differences. Before we proceed, however, it is important to reiterate a basic principle that underlies the algorithm in [8] and is also applicable here:

Basic Principle: If a node $v \in F$ can be assigned a direction, say d, that does not conflict with any direction that could possibly be assigned to the rest of the nodes in F, then it is sufficient to just search for a valid assignment for the nodes in $F - \{v\}$ and assign the direction d to v.

A justification for the above claim is quite obvious; however, the underlying principle is very useful.

A. Efficient Algorithms for Special Cases

Let us consider the following four special cases:

Case 1: The line segments assigned to the nodes in F can be along only two directions, and the permitted directions are opposite to each other, e.g., {*Left, Right*} (see Fig. 2(a); note that in the figures, a permissible direction is shown by drawing a line along the corresponding side). This case corresponds to the situation where spare processors are available only along two opposite sides of the array.

Case 2: The line segments assigned to the nodes in F can be along only two directions, and the permitted directions are at right angles, e.g., {*Left, Down*} [see Fig. 3(a)].

Case 3: The line segments assigned to the nodes in F can be along only three directions, e.g., {*Left, Right, Up*} [see Fig. 3(b)].

Case 4: The nodes in the grid are partitioned into three distinct regions, as shown in Fig. 3(c). In region A, there are three permissible directions (e.g., {*Left, Up, Right*}), in region B, there are two permissible directions (e.g., {*Left, Right*}), and in region C, there are three permissible directions (e.g., {*Left, Down, Right*}).

Because of the differences between the cases where $m = 1$ and m arbitrary (as noted before), the algorithms to be developed for the special Cases 1 and 3 are much more involved than those in [8]. The algorithms for the special Cases 2 and 4 are going to be rather straightforward extensions of those for $m = 1$.

Lemma 1: There is a linear time algorithm for determining a valid assignment for Case 1.

Proof: Let us consider, without loss of generality, the case when the permissible directions are *Left* and *Right*; the main objective here is to assign directions such that near-miss situations are avoided.

Suppose that in a certain row i there are k nodes from F. If k is greater than $2m$, which is the total number of spare processors on the two sides, then it is obvious that there is no valid assignment. If k is less or equal than $2m$, then there are at most k possible ways that the F nodes of row i can get assigned. We shall represent $w_{i,j}$ as the assignment where $(k - j)$ leftmost nodes are assigned direction *Left* and the j rightmost nodes are assigned direction *Right*. Note that, if $(k - j) > m$ or $j > m$ then $w_{i,j}$ is not a valid assignment (otherwise, more than m lines will overlap along the ith row); this also implies that the total number of possible assignments of the nodes in any row (represented as m_i) is $\leq m$.

An algorithm for determining a valid assignment can now be described as follows:

1) Construct a layered graph $G(V, E)$ the following way:
   - V has a layer for each row i; each such layer has m_i nodes, where each node corresponds to one of the distinct possible assignments of the nodes (belonging to F) in row i [see Fig. 2(b)]. In other words, a node $v_{i,l}$ in layer i of the graph G corresponds to a valid assignment, say $w_{i,l}$, of the nodes of F in row i.
   - Edges are *only* defined between nodes in adjacent layers as follows: there is an edge between a node $v_{i,l}$ (a node in layer i) and a node $v_{i+1,n}$ [a node in layer $(i+1)$] iff corresponding assignments $w_{i,l}$ and $w_{i+1,n}$ do not create a near-miss situation.

   It is obvious from the construction of $G(V, E)$, that there is a valid assignment of directions (without near-miss situations) to the nodes, if and only if there is a path from some node in layer 1 (corresponding to the bottom row) to some other node in layer N (corresponding to the topmost row); the nodes lying on such a path determine the line assignments for the nodes of F in every row.

   Following steps illustrate one way of finding such a path and the corresponding valid assignments.
2) Consider the nodes of the first layer. Mark all the nodes in the first layer as well as the edges that are incident to the marked nodes.

   Then starting with layer $i = 2$ do the following:
   - Mark every node $v_{i,k}$ for which there exist a marked edge incident to it.
   - Mark all the edges that connect marked nodes of layer i with nodes of layer $(i + 1)$.
   - Go to the next layer, $(i + 1)$.

   If the procedure above ends with one or more marked nodes in layer N, then there is a valid assignment. Otherwise, there is not a valid assignment. [See Fig. 2(c).]
3) To determine the valid assignments do the following:

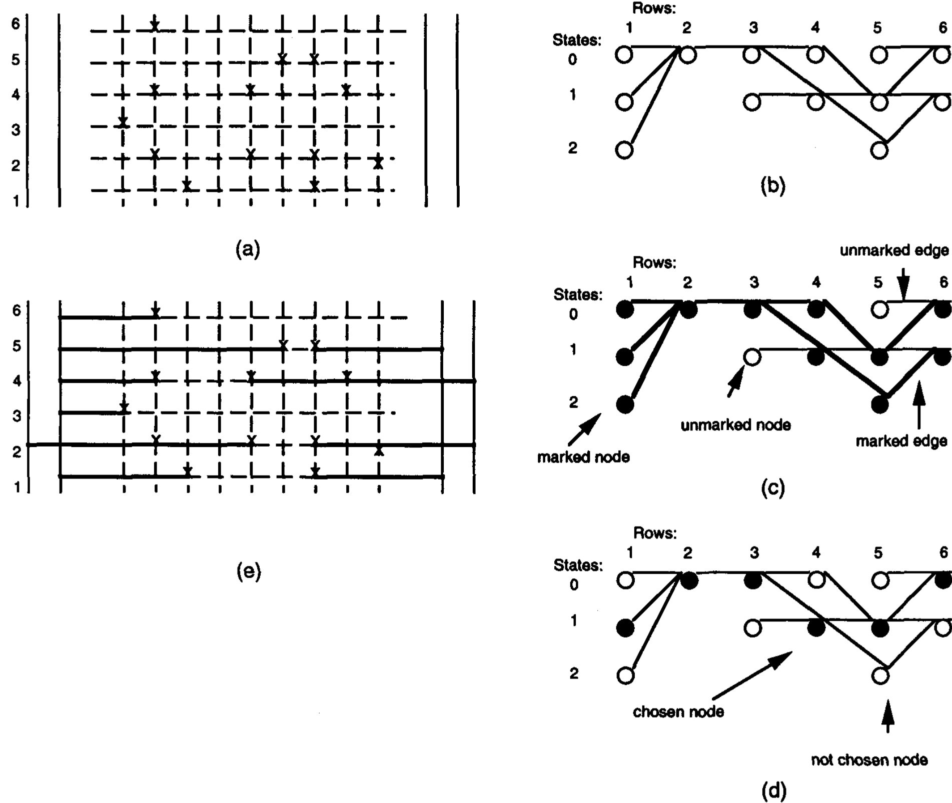


Fig. 2. Reconfiguration for special cases: Case 1.

- Start with layer N and process the layers backwards. That is, pick a marked node in layer N, say $v_{N,l}$, identify it as *chosen*, and assign the nodes of row N the directions corresponding to $w_{N,l}$.

  Then starting with layer $i = N - 1$ do the following:
- Pick a marked node $v_{i,l}$ in layer i that is incident to the *chosen* node in layer $(i+1)$ and indicate it as *chosen*. Assign the nodes of row i the assignment $w_{i,l}$ that corresponds to the $v_{i,l}$ vertex.
- Go to layer $i - 1$.

  [See Fig. 2(c), (d), and (e).]

The above algorithm visits every column of the graph twice and checks its nodes one at a time, thus its complexity is $O(m|F|)$. □

The following lemma will be useful for developing algorithms for Case 3:

Lemma 2: For Special Case 1, there is a linear time algorithm for determining the maximum number of consecutive rows in the grid, starting at the bottom row, such that all nodes in these rows have valid assignments.

Proof: The algorithm can be presented as follows:

1) Apply the first and the second step of the algorithm presented in Lemma 1.
2) Let $(k + 1)$ be the first layer in G, that has no marked nodes. Thus, nodes in row $(k+1)$ cannot be given a valid assignment if the nodes in the all the rows below are given valid assignments. Hence, k is the desired number.
3) Apply step 3 of the algorithm presented in Lemma 1 starting with $i = k$ to determine a valid assignment.

Lemma 3: There is a linear time algorithm for determining a valid assignment for Case 2.

Proof: Without loss of generality let us assume that the permitted directions are *Left* and *Down* [see Fig. 3(a)]. An algorithm for determining a valid assignment can be described as follows (it is easy to see that no near-miss situation can arise in this case):

Sequentially examine the rows of the grid starting with the bottommost row. For every node $v \in F$ in the row try to assign the direction *Down*; note that m lines are allowed to overlap along a row or column. Thus, there are two cases:

1) All nodes in the row can be assigned the direction *Down*; in which case go to the upper row and repeat the procedure.
2) There is a node $x \in F$ that cannot be assigned the direction *Down*; this can happen only if there are m other nodes in F that are in the same column as x but in rows that have already been examined. Try to assign the direction *Left* to the node x, if it cannot be done then there is no valid assignment.

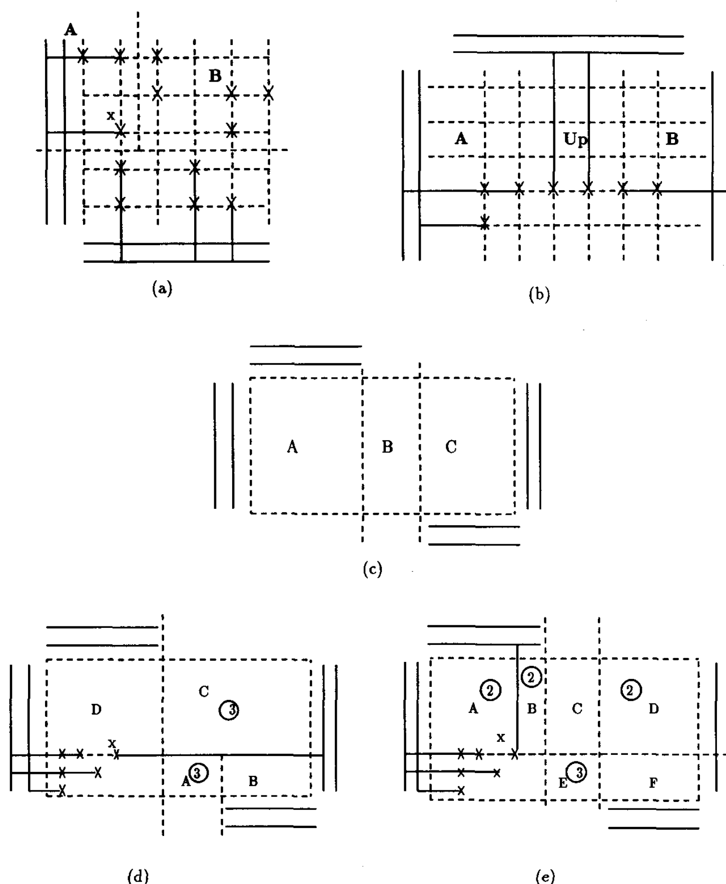


Fig. 3. Reconfiguration for special cases: Case 2, Case 3, Case 4.

If x can be assigned the direction *Left*, then as shown in Fig. 3(a), the unexamined region of the grid is divided into two distinct regions. Assign all the nodes in the left top region the direction *Left*; if it is not possible to do so then there is no valid assignment. If the nodes in this region are all successfully assigned the direction *Left*, then go to the bottom row in the region B and repeat the procedure described till now.

The above algorithm visits every node exactly once; hence it is of linear complexity. The justification for the above algorithm can be summarized as follows:

1) Consider the nodes belonging to F in the bottom row: if all these nodes are assigned the direction *Down*, then the possible assignments to the nodes in the rest of the grid are not affected. Hence, by the basic principle, one can assign the direction *Down* to the nodes in the bottom row and search for valid assignments for the nodes belonging to F in the rest of the grid. The same argument holds for the rest of the rows.
2) Now, a node x cannot be assigned the direction *Down* if and only if there are m nodes, say $x_1, x_2 \cdots x_m$, that are in the same column as x, but in a lower row. One is now left with no alternative but to assign the direction Left to x. If however, there are m other nodes of F that are to the left of x, then assigning *Left* to x is not valid and hence, there is not a valid assignment. If *Left* can be assigned to x then the nodes in the region A [as shown in Fig. 3(a)] cannot be assigned the direction *Down*; hence the nodes in the region A have to be assigned the direction *Left*. If however, there are more than m nodes in any row of the region A, then all these nodes cannot be simultaneously assigned the direction *Left*; in which case there does not exist a valid assignment.

At this stage of the algorithm, the unassigned nodes are only in region B, and the algorithm proceeds identically. □

Lemma 4: There is a linear time algorithm for determining a valid assignment in Case 3.

Proof: Without loss of generality, let us consider an instance of Case 3 as shown in Fig. 3(b), where the permitted directions are *Left*, *Right*, and *Up*.

An algorithm for finding a valid assignment can be described as follows:

Step 1: Start with the bottom row and apply the algorithm presented in Lemma 2 to determine the largest number of rows k for which a valid assignment according to Case 1 (i.e., assigning only directions *Left* and *Right*) can be found. If $k = N$, then a valid assignment can be found for every node without

assigning the direction *Up* to any of the nodes. If $k < N$, then go to the next step.

Appendix A shows that assigning direction *Up* to any of the nodes of the bottommost k rows will introduce more constraints to the assignment of the rest of the nodes. So assigning all the nodes of the first k rows directions *Left* and *Right*, and assigning direction Up to certain nodes of the $(k+1)$th row is going to be an optimum assignment for the whole array.

We define a *candidate-assignment of any row* as a possible way to assign directions *Left, Right, and Up* to the nodes of the row, i.e., a candidate-assignment specifies the following: 1) the consecutive nodes ($\leq m$), starting from the left end, to be assigned direction *Left*, 2) the consecutive nodes ($\leq m$), starting from the right end, to be assigned direction *Right*, and 3) the remaining nodes in the middle, to be assigned direction *Up*.

We are going to consider all the candidate-assignments of row $(k+1)$. For each of these assignments we are going to check whether it is compatible with the assignment of the rest of the array, meaning: 1) whether valid assignments can be found for the first k rows of the array (according to Case 1) that do not create a near-miss with the given assignment of row k and 2) whether a valid assignment can be found for the top rows of the array that do not interfere with the given assignment of row $(k+1)$. If we do not succeed in finding such a valid assignment for the rest of the array, we try another candidate-assignment of row $(k+1)$ until a valid assignment for the whole array is found or until we exhaust all the candidate-assignments of row $(k+1)$. Steps 2 and 3 describe the way of handling such a search.

Step 2: Consider a particular candidate-assignment of the nodes in the $(k+1)$th row and construct the graph presented in the algorithm for Case 1 (Step 1) for the first $(k+1)$ rows. In the $(k+1)$th layer of this graph include only the node that corresponds to the chosen candidate-assignment of row $(k+1)$, where the nodes that are assigned direction Up are ignored. Apply steps 2 and 3 of the same algorithm to find a valid assignment for the first $k+1$ rows. If one does not succeed then try another candidate-assignment of row $(k+1)$. If one succeeds, then go to the next step.

Step 3: Find a valid assignment for the rightmost and the leftmost top subarrays A and B shown in Fig. 3(b) according to Case 2 (i.e., assigning only directions *Left-Up* and *Right-Up*, respectively). Assign direction Up for all the nodes that have not been assigned a direction yet. If you do not succeed then find another possible assignment of row $(k+1)$ and go to Step 2. If all the possible assignments of row $(k+1)$ are exhausted then there is no valid assignment.

The algorithm presented above is $O(m^2|F|)$.

We make the following comments on the proof of the algorithm presented above.

- Once the assignment of the first k rows using directions *Right* and *Left* is optimum and there cannot be found such an assignment for the first $(k+1)$ rows, then some of the nodes of row $(k+1)$ have to get assigned direction Up (see Appendix A).
- Note that a candidate-assignment can be characterized by 1) a *left end*, such that all nodes to the left of it are assigned direction *Left*, and 2) a *right end*, such that all nodes to the right of it are assigned direction *Right*. The nodes in the middle are then assigned direction *Up*. Since, at most m lines can overlap along any row, it is easy to observe that the total number of candidate-assignments is equal to the number of ways one can pick the left and right ends (from among the leftmost m nodes and the rightmost m nodes). Hence, the total number of candidate-assignments of a row $\leq m^2$.
- Subarrays A and B mentioned in Step 3 have only two assignment possibilities: $Up\text{-}Left$ and $Up\text{-}Right$, respectively, and the rest of the unassigned nodes have Up as the only possible assignment.

A more efficient algorithm [time complexity of $O(m|F|)$] for Case 3, that does not require exhaustive search of all the candidate-assignments of row $(k+1)$, can also be designed. However, the description of the algorithm is more complicated and is presented in Appendix B. □

Lemma 5: There is a linear time algorithm for determining a valid assignment in Case 4.

Proof: Let us consider an instance of Case 4 as shown in Fig. 3(c). An algorithm for determining valid assignments can be outlined as follows:

Sequentially examine the rows of the grid, starting with the bottom row, and consider the nodes of F that are in region A only.

If the row under consideration has m nodes in the region A or less, then assign them the direction *Left* and go to the next upper row. If the row has no nodes from F, then also go to the next upper row and repeat the procedure.

If the row has more than m nodes, then the rightmost node, say x, has at most two directions (namely, *Right* and *Up*) that can be assigned to it. The algorithm checks for valid assignments by first assigning the direction *Right* to node x, and then assigning the direction *Up* as follows:

1) Assign x the direction *Right*; Fig. 3(d) shows the partitioning of the grid under this assignment. The unassigned parts of the grid get partitioned into four different regions and each such region can be labeled as one of the cases that we have already discussed. For example, the region A has two permissible directions that are opposite to each other and hence can be searched for valid assignments using the algorithm discussed in Lemma 1. However, consider the region D and C together; the whole region can be treated as a restricted version of Case 3 discussed in Lemma 4. It is restricted in the sense that for the nodes of F that are in the region C, the

direction *Up* is disallowed. We can easily accommodate such restrictions by just adding m additional rows in the top of the array that have faulty PE's all along the columns of region C. Generaly all restrictive cases can be handled using the idea of adding rows/columns that have faulty PE's in the columns/rows where there is a restriction. Hence, the combined region D and C can be searched for valid assignments by using the algorithm outlined in Lemma 4. Similarly, one can again use the algorithm of Lemma 4 to search for valid assignments in the combined region A and B [see Fig. 3(d)].

2) If the result of the previous search is negative then assign x the direction *Up*; Fig. 3(e) shows the resulting partitioning of the grid. The grid gets partitioned into six regions and each region can be labeled by the case it corresponds to. For example, the region A has two permissible directions that are at right angles; hence it corresponds to Case 2. Similarly, the region C has only one permissible direction and hence all nodes in the region are assigned the direction *Right*.

   Search the regions sequentially in the following order: A, B, C, D and E, F (combined) using the corresponding algorithms [as shown in Fig. 3(e)].

Note that if none of the rows in the region A of Fig. 3(c) has more than m nodes belonging to F, then one can assign the direction *Left* to the nodes in the region A. One can then search for valid assignments in the regions B and C of Fig. 3(c), by combining the two regions and then treating the combined region as a restricted version of Case 3.

In the above algorithm each node is visited at most twice and hence the algorithm is again of linear complexity. □

Remarks: 1) The algorithms developed for the special cases are also valid in the case where some of the assignment directions are not permitted (also see [6] and [8]). 2) Our algorithms can also handle the cases where some of the spare PE's are faulty. This case is equivalent to the restricted assignment case mentioned in remark 1.

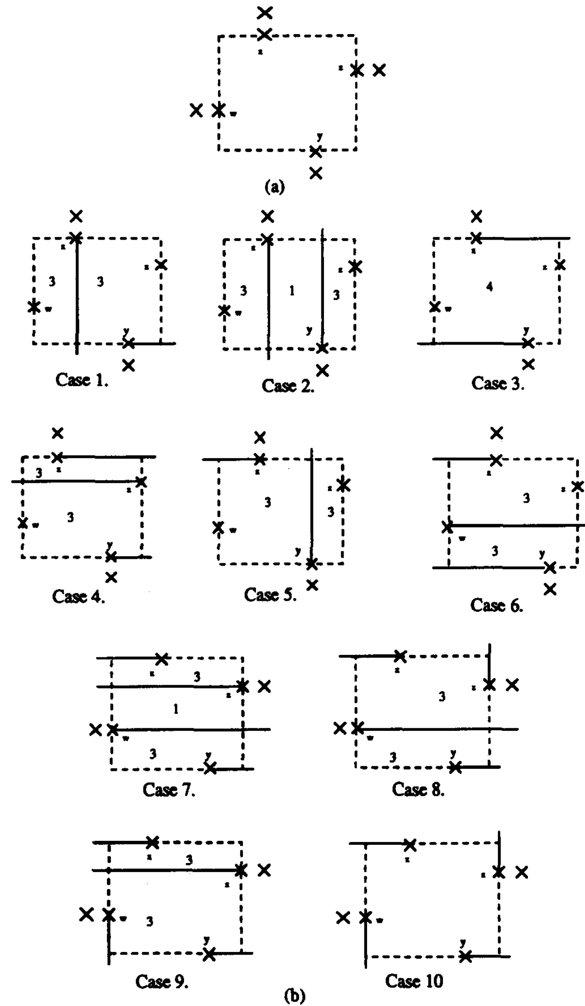


Fig. 4. Reconfiguration for the general case.

B. Efficient Algorithms for the General Case

The algorithm for the general case where all the four directions are permitted can be described as a layer peeling algorithm. It starts with the outermost rows and columns of the grid and determines *valid directions* for the nodes of F that are in these outer layers; it then performs the same operations on the inner layers. The algorithm can be discussed in two parts as follows.

Part 1: In the first part of the algorithm, one attempts to determine valid directions using the Basic Principle stated in the beginning, and it can be described as follows:

1) Sequentially examine the columns of the grid starting with the leftmost column. Try to assign the direction *Left* to every node of F that is on the column under consideration. If all the nodes can be successfully assigned the direction *Left*, then go to the next column. (Note that a node cannot be assigned direction left only if there are m nodes on its left and on the same row.) If there is a node that cannot be assigned the direction *Left*, then go to the next step.
2) Examine sequentially the rows of the reduced grid (i.e., the portion of the grid that is unassigned) starting from the top, and assign, as long as it is possible, the direction *Up* to all of its nodes.
3) Examine sequentially the columns of the reduced grid, starting from the right, and assign, as long as it is possible, the direction *Right* to all of its nodes.
4) Examine sequentially the rows of the reduced grid, starting from the bottom, and assign, as long as it is possible, the direction *Down* to all of its nodes.
5) Repeat Steps 1 to 4 for the unassigned part of the grid until the reduced grid is of the form shown in Fig. 4(a). In particular, each of the outermost row and column of the reduced grid should have at least one node that is blocked on the outside.

Part 2: The next part of the algorithm determines valid directions for the nodes on the outermost rows and columns

of the grid shown in Fig. 4(a). The objective is to show that we can determine such valid directions in linear time.

The idea is to enumerate all the possible ways the nodes x, y, ,w, and z can be assigned directions and then use the algorithms for the special cases to check for valid assignments. A complete algorithm for a systematic search of a valid assignment for the nodes in the outer layers of the grid shown in Fig. 4(b) can be summarized as follows:

1) Enumerate all the possible assignments as shown in Fig. 4(b); there are at most ten possible such assignments.
2) Use the algorithms developed for special cases to search for valid assignments in each of the first nine configurations, which are illustrated in Fig. 4(b); the partitioning of the regions and the special cases they correspond to are also shown in the figure. If any of the searches finds a valid assignment for every node in the reduced grid then there is a successful solution to Problem 1.
3) If every search in Step 2 fails then assign directions to nodes x, y, w, and z that are shown in Case 10 of Fig. 4(b), and go to the inner portion of the grid (i.e., peel off the outermost layer) and repeat Parts 1 and 2 for the new reduced grid obtained after peeling off the outer layers.

   The idea is the following: in Step 2, we have checked all but one possible assignment to the nodes in the outermost layer of the reduced grid. If none of these cases leads to a valid assignment for nodes of F that are in the grid, then for a valid assignment to exist the last possible assignment [i.e., Case 10 in Fig. 4(b)] is the only candidate.

Note that if the outermost rows and columns in the reduced grid [Fig. 4(a)] have more than one node, then some of the cases in Fig. 4(b) will not be feasible and the search for valid assignments will be even simpler.

Theorem 1: There is a quadratic time algorithm ($O(m|F|^2)$) for determining if there exists a valid assignment for Problem 1.

Proof: Follows directly from the description of Parts 1 and 2 of the algorithm. To decide on valid directions for nodes lying on the outermost layers of the grid, one needs at most $O(|F|)$ time. Since there are at most $m|F|$ such layers in the grid, the complexity of the algorithm is $O(m|F|^2)$. □

III. Concluding Remarks

In this paper we presented a polynomial time algorithm for solving the combinatorial problem that underlies the reconfiguration issues in the $m\frac{1}{2}$-track-m-spare model, for any arbitrary m. More precisely we solved the following combinatorial problem: given a set of points in a two-dimensional grid, find a set of nonintersecting straight lines such that 1) every line starts at a point and connects it to one of the boundaries of the grid, 2) there are no more than m lines overlapping in any row or column of the grid, and 3) there are no near-miss situations (as defined in the Introduction). The time complexity of our algorithm is $O(m|F|^2)$, where $|F|$ is the number of faulty processors. One open issue is to design a more efficient algorithm that improves the time-complexity of the algorithm presented in this paper. Another open issue is to derive a polynomial time algorithm for the following problem when m is arbitrary: If in Problem 1, a set of straight lines covering all the nodes in F cannot be found, then determine a set of lines such that ***a maximum number of nodes in F*** is covered. The corresponding problem for the special case of $m = 1$ has been solved in [2].

Appendix

Appendix A

We are going to prove here that in the algorithm developed for Case 3, one does not need to assign direction *Up* to any node in the first k rows. Suppose that the algorithm developed for Case 3 fails, then we will show that by assigning direction *Up* to any node in the first k rows one cannot obtain a valid assignment.

Let us make the following observations about our algorithm:

1) Assignment of direction *Up* to the nodes in row $(k+1)$ is the *only way* that assignment of directions to the nodes in the region above row $(k+1)$ is restricted.
2) We define *the best candidate-assignment* of row $(k+1)$ as the candidate-assignment that has the largest possible number of nodes assigned directions *Left* and *Right* and is still compatible with the assignment of the first k rows (note that there exists such a candidate assignment, since the left assignment of the nodes of row $(k+1)$ does not interfere with the right assignment of the nodes of row $(k+1)$ so one can maximize them both independently).
3) Starting from the best candidate-assignment one could argue that assigning direction *Up* to some node in the first k rows, say z, (instead of direction *Left* or *Right* that has been assigned so far) might enable some extra nodes of row $(k+1)$ to get assigned directions *Right* or *Left* (which was impossible before). This might remove some of the restrictions imposed by row $(k+1)$ to the assignment of the nodes of the rows above row $(k+1)$.
4) We are going to prove that the best candidate assignment cannot be improved by the assignment of direction *Up* to any node z in the first k rows.

Consider the best candidate assignment of row $(k+1)$ as shown in Fig. 5 and let y_l and y_t be its left and right end, respectively.

Suppose that there is a node, say z, in the first k rows whose assignment is changed to direction *Up* (note that in Step 1 of the algorithm for Case 3, z was originally assigned direction *Left* or *Right*). Then we consider the following cases:

Case 1: Node z is in subarray A. Assigning z direction Up would force some of the nodes $y_1 \cdots y_l$ to get assigned direction Up (in order to avoid intersection). Obviously, this will not increase the number of nodes that are assigned direction $Left$. It will also not improve the number of nodes that are assigned direction $Right$ either because:

1) If there are already m nodes to its right, then of course y_{t-1} cannot get assigned direction $Right$.
2) If y_{t-1} cannot get assigned direction $Right$ because of near-miss restrictions, then it can be easily shown that

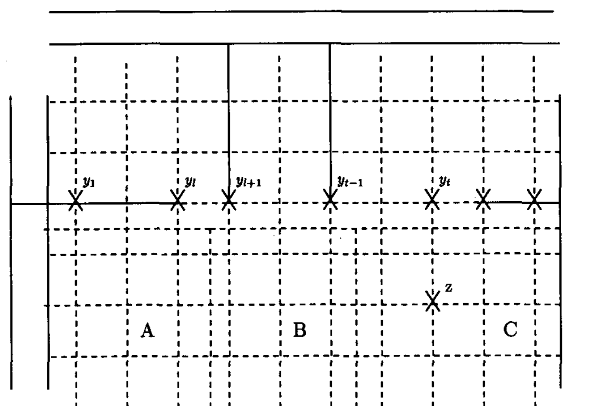


Fig. 5. Special case 3.

direction $Right$ cannot be made feasible by changing the assignment of a node that is to the left of y_{t-1}.

Case 2: Node z is in subarray C. A similar argument holds for this case as well.

Case 3: Node z is in subarray B. If node z gets assigned direction Up then the right and the left assignments of the above nodes might become impossible due to intersection. If y_{l+1} and/or y_{t-1} cannot get assigned directions $Left$ and $Right$, respectively, due to the presence of m nodes on the left/right, then again the Up assignment of z cannot help. If y_{t-1} and y_{l+1} cannot get assigned directions $Right$ and $Left$ due to near-miss situations then nodes in B cannot interfere with that because they are on the left of y_{t-1} and on the right of y_{l+1}.

So we have proved that assigning direction Up to node z cannot reduce the restrictions for the assignment of the nodes in the rows above $(k+1)$. □

APPENDIX B

Another algorithm for finding a valid assignment for Case 3 that does not require a search of all the candidate-assignments of row $(k+1)$, and is of time complexity $O(mF)$ will be presented here. In this algorithm we try to assign directions *Left* and *Right* to as many of the nodes of row $(k+1)$ as possible. In other words we try to find the best candidate-assignment (defined in the previous section) of row $(k+1)$. So, starting from a random assignment of the first k rows [which might not be compatible with the best candidate assignment of row $(k+1)$] we do systematic changes in the assignment of the k first rows, so as to make it possible to assign directions *Left* and *Right* to as many nodes of row $(k+1)$ as possible.

The algorithm can be described as follows:

Step 1: Apply the algorithm presented in Lemma 2 to determine the largest number of rows k for which a valid assignment according to Case 1 can be found. If $k = N$ a valid assignment has been found for this case as well. In $k < N$ then go to the next step.

Step 2: Consider now row $(k+1)$. Starting from the left most node of row $(k+1)$ in F, assign direction *Left* until you reach a node z that cannot be assigned direction *Left* . Starting from the right to the left assign direction *Right* until you reach a node r that cannot be assigned direction *Right.* Name the nodes of row $(k+1)$ from z to r $y_1, y_2, \cdots, y_p$.

Step 3: Starting with node $y_i = y_1$ and the row $l = k$ do the following:

a) Assign node y_i direction *Left.* If this cannot be done because there are m nodes on y_i's left, then assign y_i direction *Up* and go to step 4.

b) For the l row do the following:

1) Starting from the left of row l assign direction *Left* to every node, until the near-miss between row l and $l+1$ is eliminated.

2) If the step above cannot be completed because there are more than m nodes that need to get assigned direction *Left* in order to eliminate near-miss, then disregard the changes in the assignments in the last i iteration, assign y_i direction *Up* and go to step 4.

3) If there is no near-miss between rows l and $(l+1)$ increase i and repeat step 3.

4) Else increase l and go to Step 3.b.

Step 4: Starting with node $y_j = y_p$ and the row $l = k$ do the following:

a) Assign node y_i direction *Right.* If this cannot be done because there are m nodes on y_i's right, then assign y_i direction *Up* and go to step 5.

b) For the l row do the following:

1) Starting from the right of row l assign direction $Right$ to every node, until the near-miss between row l and $l+1$ is eliminated.

2) If the step above cannot be completed because there are more than m nodes that need to get assigned direction *Right* in order to eliminate the near-miss, then disregard the changes in the assignments in the last j iteration, assign y_j direction *Up* and go to step 5.

3) If there is no near-miss between rows l and $(l+1)$ decrease j and repeat step 4.

4) Else increase l and go to Step 5.b.

Step 5: Assign direction *Up* to all nodes $y_i \cdots y_j$.

Step 6: Find a valid assignment for the rightmost and the leftmost top subarrays A and B as shown in Fig. 3(b) as in Case 2. If not possible signal failure. Assign direction *Up* for all the nodes in F that have not been assigned a direction yet. If not possible signal failure.

On the justification of the algorithm above, we shall make the following comments:

1) It is not possible to find a valid assignment where the nodes of the $(k+1)$ rows are assigned all directions $Left$ or $Right$.

2) It is optimum to try to assign directions *Left* and *Right* to as many nodes of row $(k+1)$ as possible. This kind of assignment introduces the least number of conflicts with the assignments of the rows above. Furthermore, the nodes of row $(k+1)$ that get assigned direction *Left* do not interfere at all with the nodes of row $(k+1)$ that get assigned direction $Right$. So trying to maximize the number of nodes of $(k+1)$ row that get assigned direction *Left* does not have any effect to our effort to maximize the number of nodes of row $(k+1)$ that get assigned direction *Right*.
3) When the *Left* (or *Right*) assignment of a node creates a near-miss with a neighboring row, then its *Right* (or *Left*) assignment CANNOT create a near-miss with the same neighboring row. As a result of that when the *Left* (*Right*) assignment of a node z of row $(k+1)$ creates a near-miss situation between rows k and $(k+1)$ then it is guaranteed that the *Right* (*Left*) assignment of node z would not near-miss with any node of row k. Furthermore, if every node on the left (right) of z is assigned direction *Left* (*Right*), *Right* (*Left*) direction is no more valid for z because there are m nodes or more on the right (left); (if such an assignment was valid then one could have assigned all the nodes to the right (left) of node z direction *Right* (*Left*) which would be contradicting comment 1 stated above).

Having these comments in mind we shall outline the proof of the correctness of the algorithm presented above.

- Having found already in Step 1 the largest number k of bottom rows that can get reconfigured, assigning just directions *Left* or *Right*, in Step 2 we try to assign as many nodes of row $(k+1)$ as possible directions *Left* or *Right* which is an optimum approach as indicated in comment 2 above.
- Coming to the ith iteration of step 3 we try to assign direction *Left* to node y_i of row $(k+1)$. If this is not possible because there are m nodes on y_i's left, then there is not other alternative for y_i than direction Up (Step 3.a). Note that according to comment 4 above direction *Right* is not valid for y_i.
- If y_i cannot be assigned direction *Left* because of a near-miss with some node(s) on its left in row k already being assigned direction *Right*, we shall try to change the assignment of row k to eliminate the near-miss. To eliminate the near-miss one could assign direction Up to some of the nodes of row k, but that would force y_i to get assigned direction Up as well to avoid intersection. This would be a more restrictive assignment than assigning node y_i direction Up from the beginning. So the only way to eliminate the near-misses without introducing more restrictions in the assignment of the rest of the nodes is to change the assignment of the leftmost nodes of row k from *Right* to *Left* (Step 3.b.1). We carry through this procedure in the l iteration of step 3 making the necessary changes in the assignment of row l to eliminate the near-miss that the changes in the $l-1$ row introduced, until we reach a node blocked on the left in which case assigning y_i direction *Left* is impossible (Step 3.b.2), or until no more near-miss situations appear, in which case we consider the next node of row $(k+1)$ (Step 3.b.1).
- Once we have assigned a node y_i of row $(k+1)$ direction Up we go to step 4 which is the same as Step 3 starting now from the right to the left of row k, until we find the first node y_j that is assigned direction Up. The only possible assignment for the nodes of row $(k+1)$ between y_i and y_j is Up (Step 5).

□

ACKNOWLEDGMENT

We would like to thank the referees and J. A. B. Fortes for their helpful comments.

REFERENCES

[1] Y. Birk and J. B. Lotspiech, "On finding non-intersecting straight-line interconnections of grid points to the boundary," Tech. rep., Tech. Rep. RJ 7217 (67984), IBM, Almaden Research Center, San Jose, CA, Dec. 1989.

[2] J. Bruck and V. Roychowdhury, "How to play bowling in parallel on the grid?" Tech. rep., Tech. Rep. RJ 7209 (67688), IBM, Almaden Research Center, San Jose, CA, Dec. 1989.

[3] M. Chean and J. A. B. Fortes, "The full-use-of-suitable-spares (fuss) approach to hardware reconfiguration for fault-tolerant processor arrays," *IEEE Trans. Comput.*, vol. 39, pp. 564–571, Apr. 1990.

[4] J. S. N. Jean, "Fault-tolerant array processors using n-and-half-track switches," in *Proc. Int. Conf. Appl. Specific Array Processors,* Sept. 1990.

[5] J. S. N. Jean, H. C. Fu, and S. Y. Kung, "Yield enhancement for WSI array processors using two-and-half-track switches," in *Proc. Int. Conf. Wafer Scale Integration,* San Francsco, CA, Jan. 1990, pp. 243–250.

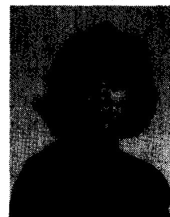
[6] S. Y. Kung, S. N. Jean, and C. W. Chang, "Fault-tolerant array processors using single-track switches," *IEEE Trans. Comput.,* vol. 38, no. 4, pp. 501–514, Apr. 1989.

[7] F. Lombardi, M. G. Sami, and R. Stefanelli, "Reconfiguration of VLSI arrays by covering," *IEEE Trans. Comput.-Aided Design IC Syst.,* vol. 8, no. 9, pp. 952–965, Sept. 1989.

[8] V. P. Roychowdhury, J. Bruck, and T. Kailath, "Efficient algorithms for reconfiguration in VLSI/WSI arrays," *IEEE Trans. Comput.,* Special Issue on Fault tolerant Computing, vol. 39, no. 4, pp. 480–489, Apr. 1990.

[9] T. Varvarigou, V. Roychowdhury, and T. Kailath, "New algorithms for reconfiguring VLSI/WSI arrays," *J. VLSI Signal Processing,* vol. 3, no. 4, pp. 329–344, Oct. 1991.

[10] ——, "Reconfiguring arrays using multiple-track models: The 3-track-1-spare approach," *IEEE Trans. Comput.,* to be published.



Theodora A. Varvarigou was born in Athens, Greece, in 1966. She received the B.Tech. degree from the National Technical University of Athens, Athens, Greece, in 1988, the M.S. and Ph.D. degrees from Stanford University, Stanford, CA, in 1989 and 1991, respectively.

She is currently working at AT&T Bell Laboratories, Holmdel, NJ. Her research interests include distributed telecommunication systems, multimedia and multiparty communications, parallel algorithms and architectures, fault-tolerant computation and parallel scheduling on multiprocessor systems.

Vwani P. Roychowdhury received the B.Tech. degree from the Indian Institute of Technology, Kanpur, India, and the Ph.D. degree from Stanford University, Stanford, CA, in 1982 and 1988, respectively, all in electrical engineering.

From September 1989 to August 1991 he was a Research Associate in the Department of Electrical Engineering at Stanford University. He is currently an Assistant Professor in the Electrical Engineering Department, Purdue University. His research interests include parallel algorithms and architectures, design and analysis of neural networks, special purpose computing arrays and VLSI design, and fault-tolerant computation.

Thomas Kailath (S'57–M'62–F'70) was educated in Poona, India, and at the Massachusetts Institute of Technology (S.M., 1959; Sc.D., 1961).

From October 1961 to December 1962, he worked at the Jet Propulsion Laboratories, Pasadena, CA, where he also taught part-time at the California Institute of Technology. He then came to Stanford University as an Associate Professor of Electrical Engineering. He served as Director of the Information Systems Laboratory from 1971 through 1980, as Associate Department Chairman from 1981 to 1987, and currently holds the Hitachi America Professorship in Engineering. He has held short term appointments at several institutions around the world. He has worked in a number of areas including information theory, communications, computation, control, signal processing, VLSI design, statistics, linear algebra, and operator theory; his recent interests include applications of signal processing, computation and control to problems in semiconductor manufacturing and wireless communications. He is the author of *Linear Systems* (Englewood Cliffs, NJ: Prentice-Hall, 1980), and *Lectures on Wiener and Kalman Filtering* (Berlin, Germany: Springer-Verlag, 1981).

Dr. Kailath has held Guggenheim, Churchill, and Royal Society fellowships, among others, and received awards from the IEEE Information Theory Society and the American Control Council, in addition to the Technical Achievement and Society Awards of the IEEE Signal Processing Society. He served as President of the IEEE Information Theory Society in 1975, and has been awarded honorary doctorates by Linköping University, Sweden, and by Strathclyde University, Scotland. He is a Fellow of the Institute of Mathematical Statistics and is a member of the National Academy of Engineering.