Efficient Algorithms for Reconfiguration in VLSI/WSI Arrays

VWANI P. ROYCHOWDHURY, JEHOSHUA BRUCK, MEMBER, IEEE, AND THOMAS KAILATH, FELLOW, IEEE

Abstract—**This paper deals with the issue of developing efficient algorithms for reconfiguring processor arrays in the presence of faulty processors and fixed hardware resources. The models discussed in this paper consist of a set of identical processors embedded in a flexible interconnection structure that is configured in the form of a *rectangular grid*. We first consider an array grid model based on single-track switches for which simulations performed by previous researchers have shown that considerable enhancement in yield can be achieved by reconfiguring arrays according to a set of conditions that can be formally stated in the form of a so-called *reconfigurability theorem*. However, the important issue of developing efficient algorithms for determining whether the conditions in the reconfigurability theorem are met has not been resolved, and the algorithms proposed in the literature to do so are of exponential complexity. In this paper, we resolve this open problem by proposing an efficient *polynomial* time algorithm for determining feasible reconfigurations for an array with a given distribution of faulty processors. In the process, we also show that the set of conditions in the reconfigurability theorem is not necessary. Finally, we develop a polynomial time algorithm for finding feasible reconfigurations in an augmented single-track model and in array grid models with multiple-track switches.**

Index Terms—**Efficient polynomial time algorithm, fault-tolerant architecture, reconfigurable processor arrays, single-track and multiple-track models, wafer scale integration (WSI) technology.**

I. Introduction

This paper deals with the issue of developing efficient algorithms for reconfiguring processor arrays in the presence of faulty processors. Such studies can be easily motivated in the case of wafer scale integration (WSI) technology where, for example, a large number of processors, configured in the form of a grid, can be put on a single wafer. Due to yield problems, some of the processors are invariably going to be faulty. In such a case, instead of treating the whole wafer as defective, one can work around the faulty processors and reconfigure the rest in the form of a grid. Thus, reconfiguration methodologies can be viewed as possible tools to increase the effective yield of the processing technology.

Manuscript received June 30, 1989; revised November 27, 1989. V. P. Roychowdhury and T. Kailath are supported in part by the Department of the Navy, Office of Naval Research under Contract N00014-86-K-0726, the SDIO/IST, managed by the Army Research Office under Contract DAAL03-87-K-0033, and the U.S. Army Research Office under Contract DAAL03-86-K-0045.

V. P. Roychowdhury and T. Kailath are with the Information Systems Laboratory, Stanford University, Stanford, CA 94305.

J. Bruck is with IBM Almaden Research Center, San Jose, CA 95120.

IEEE Log Number 8933890.

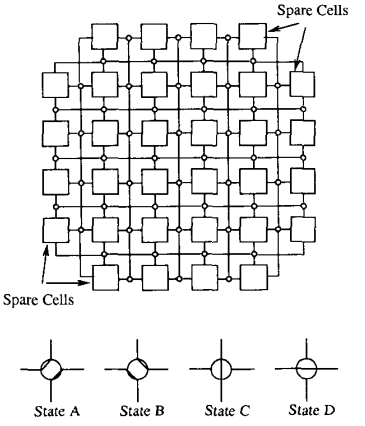


Fig. 1. The array grid model based on single-track switches, shown with the possible states of a switch.

The general model discussed here (see e.g., [5], [6], [8]–[12], [15]) consists of a set of identical processors embedded in a flexible interconnection structure that is configured in the form of a rectangular grid. Each grid line in the mesh has a fixed number of data paths that can be routed along it (i.e., the model has fixed channel width); switches can be placed at every grid point and at every location where a processor is connected to the grid. Furthermore, often the processors are divided into a set of nonspare PE's (say an $m \times n$ array) and a set of spare PE's that are distributed in a predetermined fashion. The general question asked in such models is: if some of the nonspare PE's are faulty, then can the array be reconfigured to replace the faulty PE's with some of the spare PE's? Obviously, the power of the reconfigurable architecture is determined by the available hardware resources such as the channel width, the complexity of the switches, and their distribution in the array. Although one would like to put as much hardware as possible, often it is expensive to do so.

A particularly simple but useful model is an array grid model based on single-track switches (see Fig. 1) that has been studied in [2] and [3]. It consists of an $m \times n$ array of nonspare PE's, double-row–column of spare PE's, and single-track switches; the allowed states of the switches are also shown in Fig. 1. The model is single-track in the sense that only one communication path is allowed along each horizontal/vertical channel. It is further assumed in the model that a faulty PE can be converted into a connecting element. The single-track model's advantages arises from its inherent simplicity: since data paths take up a significant amount of area

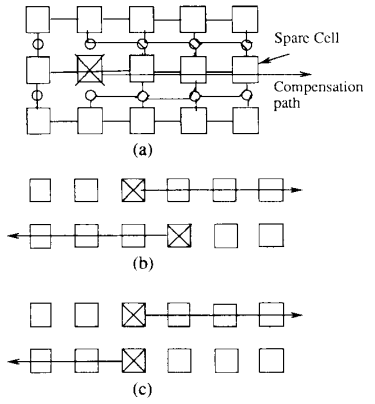


Fig. 2. Compensation paths: (a) shows the routing required by a compensation path; (b) shows a near-miss situation; (c) shows a nonnear-miss situation.

on a wafer/chip, considerable saving in area is achieved by allowing only one data path along every grid line; moreover, the simplicity of the switches makes the routing hardware more reliable. Furthermore, extensive simulations reported in [3] show that considerable enhancement in yield can be achieved by reconfiguring the array grid models with single-track switches.

We now briefly discuss the results reported in [3]. The paper derives a set of sufficient conditions (stated in the form of a so-called *reconfigurability theorem* presented below) for determining whether an array with a particular distribution of faulty processors is *reconfigurable*, where a given array is *reconfigurable* if the nonfaulty processors can be connected to form an $m \times n$ array. The sufficient conditions in the reconfigurability theorem can be 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. Fig. 2(a) shows a compensation path and the corresponding routing required for replacing a single faulty processor in the single-track model; note that the compensation path is straight and continuous. This simple concept of using straight and continuous compensation paths can be also used in the presence of multiple faulty processors and the sufficient conditions can be formally presented in the form of the so-called reconfigurability theorem (for a formal proof, see [2] and [3]).

Reconfigurability Theorem: Given an $m \times n$ array of nonspare PE's, with spare PE's along the sides, it is reconfigurable into an $m \times n$ array of healthy processors by single-track switches if 1) there exists a set of continuous and straight compensation paths covering all the faulty nonspare PE's and 2) there is neither intersection nor near-miss among the compensation paths.

A near-miss situation occurs if two compensation paths in neighboring rows (columns) overlap and are in opposite directions (see Fig. 2; note that a near-miss situation does not occur if the compensation paths overlap by one node).

In [3], an algorithm to determine valid reconfigurations that satisfy the conditions in the reconfigurability theorem is also presented. The algorithm is developed by reformulating the reconfigurability problem as a maximum independent set problem, and then adapting a well-known algorithm for determining maximum independent sets in a graph. However, the maximum independent set problem is NP-complete and the best known algorithms take exponential time; hence, the algorithm presented in [3] has exponential complexity. *The question whether efficient polynomial time algorithms exist was left as an open one. Moreover, efficient algorithms were not known even for the restricted cases where spare processors are available only along, for example, two or three sides (as opposed to on all four sides as shown in Fig. 1).*

In view of the above results, the contributions of this paper with regard to the single-track models can be summarized as follows.

- We show that the conditions in the reconfigurability theorem are *not necessary*, correcting a claim made in [1].
- We present a polynomial time algorithm (in fact, the complexity is $O(|F|^2)$, where $|F|$ is the number of faulty processors) for determining valid reconfigurations according to the sufficient conditions. Moreover, linear time algorithms for determining valid reconfigurations are developed for the restricted cases where the spare processors are not present along all four sides of the array.

We should note here that the combinatorial problem underlying the reconfigurability issues in the single-track model is by itself quite interesting. The algorithm presented in this paper appears to be the first-known polynomial time algorithm for this problem.

One can easily observe that the sufficient conditions for reconfiguration as discussed above are also valid for more general array models such as the ones with multiple tracks. One, however, hopes that for such more powerful models it should be possible to develop more general conditions to allow reconfiguration of arrays that otherwise could not be reconfigured in the single-track model. With such motivation in mind, we first consider an augmented single-track switch model as shown in Fig. 3. We show that the augmented model is more powerful than the simple single-track model: the compensation paths in the augmented model need not be straight any more and can have *bends*. In general, if one allows more data paths along every grid line (i.e., a multiple-track model), then the compensation paths can be crooked and the restriction of near misses is no longer required. Hence, a generalized sufficient condition can be stated as follows: an array grid model with multiple-track switches is reconfigurable if one can determine a set of nonintersecting compensation paths (continuous, but not necessarily straight) for the faulty PE's in the array. We show that the combinatorial problem corresponding to such a sufficient condition can be efficiently solved by reducing it to the well-known problem of determining maximum flow in networks.

The rest of the paper is organized as follows. In Section II,

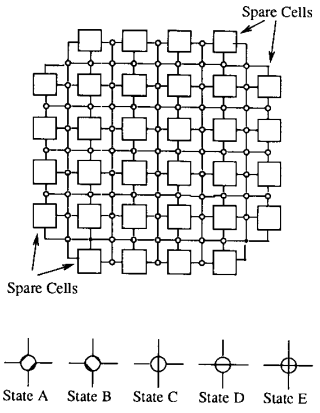


Fig. 3. An augmented single-track model.

we present polynomial time algorithms for determining valid reconfigurations in the single-track model. In Section III, we study more general models such as an augmented single-track model and multiple-track models, and present a polynomial time algorithm for searching for valid reconfigurations in such models. Section IV has some concluding remarks and in the Appendix we show that the conditions in the reconfigurability theorem are not necessary.

II. Efficient Algorithms for Reconfiguration in Single-Track Models

Although we show in the Appendix that the conditions in the reconfigurability theorem are not necessary for a valid reconfiguration to exist, extensive simulations reported in [3] show that by reconfiguring arrays according to the sufficient conditions, one can enhance the yield considerably. Now, checking whether the sufficient conditions are met is equivalent to solving the following problem: given a set of points in a two-dimensional grid, find a set of nonintersecting straight lines such that every line starts at a point and connects it to one of the boundaries of the grid, and there is no near-miss situation. We recall that a near-miss situation occurs if two lines in opposite directions overlap in adjacent rows or columns. If for the moment we relax the restriction of no near misses, then the sufficient condition reduces to the purely combinatorial problem stated in Problem 1 below.

In this section, we shall first present an efficient algorithm for solving Problem 1 and then show how the algorithm can be modified to avoid near-miss situations. The algorithm presented in this paper not only determines whether a compatible set of compensation paths exist, it also determines such a set whenever it is possible to do so. In the process, we shall develop efficient algorithms for the cases where spare processors are not available on all four sides; in fact, the algorithms for the restricted cases are only of linear (in a number of faulty processors) complexity.

The combinatorial problem can be stated as follows:

Problem 1: Let V be the set of grid points in an $m \times n$ two-dimensional rectangular grid, and let $F \subset V$. Determine a set of straight lines such that

a) each vertex $v \in F$ is assigned a straight line connecting it to one of the four boundaries of the grid

b) the straight lines are nonintersecting.

We can make the following observations:

1) If there is a row (column) in the grid that contains none of the vertices in F, then it is clear from the definition of our problem that the row (column) has no role in searching for a valid assignment of line segments. Hence, without loss of generality, we can delete such rows (columns) from the description of our problem and assume that $1 \leq m, n \leq |F|$.

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.

Definition 1: 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.

Definition 2: 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.

The basic principle underlying our algorithm will be a greedy one, and we shall try to assign valid directions to nodes of F with only a minimal search. In particular, we are always able to find a node $v \in F$ for which we can assign a valid direction efficiently (in time at most linear in $|F|$). Thus, the total complexity of the algorithm will be at most quadratic in $|F|$.

The greedy principle that will be used quite often in our algorithms is formalized by the following lemma.

Lemma 1: 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 proof of the above lemma is quite obvious; however, the underlying principle is very useful. That is, if one can identify such a node v, then one can assign it a valid direction immediately without any further search. The algorithm can then remove the node and deal with a problem of a smaller size. The algorithms that we are going to develop utilize the geometry of the grid appropriately to identify such specialized nodes in a systematic manner.

In case there are no nodes that can be assigned a direction using the above idea, then we are able to chose nodes appropriately such that they can be assigned a valid direction with only a linear search. The principles used are slightly more complicated; however, they use the structure of the grid and are also greedy in nature. The principles can be best explained by describing the individual algorithms.

A. Efficient Algorithms for Special Cases

We are going to develop our algorithm by first considering several special cases, wherein the possible directions that can be assigned to the vertices in F are restricted. We shall then use the special cases appropriately to search for valid assignments, if there are any, in the case where all the four possible directions are permitted. The four special cases are as follows.

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., $\{Left, Right\}$.

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. 4(a)]; note that in the figures, a permissible direction is shown by drawing a line along the corresponding side.

Case 3: The line segments assigned to the nodes in F can be along only three directions, e.g., $\{Left, Right, Up\}$ [see Fig. 4(b)].

Case 4: The nodes in the grid are partitioned into three distinct regions, as shown in Fig. 5(a). 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., $\{Left, Down, Right\}$).

Lemma 2: 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*. It is clear that for the case under consideration, a valid assignment exists if and only if every row contains two or less nodes belonging to F. Thus, a linear time algorithm can be designed by sequentially examining the rows from top to bottom until either a row containing three or more nodes in F is detected (in which case no valid assignment exists) or all the rows are examined. □

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 permissible directions are *Left* and *Down* [see Fig. 4(a)]. An algorithm for determining a valid assignment can be described as follows. Sequentially examine the rows of the grid starting with the bottom row. If it is possible, then for every node $v \in F$ in the row assign the direction *Down*. 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*; note that this can happen only if there is another node in F that is the same column as x but in a row that has already been examined. Try to assign the direction *Left* to the node x, if it cannot be down then there is no valid assignment.

If x can be assigned the direction *Left*, then as shown in Fig. 4(a), the unexamined region of the grid is divided into two distinct regions, namely A and B. Assign all the nodes in region A the direction *Left*; if it is not possible to do so, then

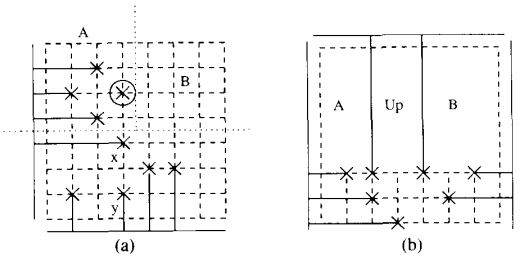


Fig. 4. Special Cases 2 (a) and 3 (b) in Section II-A.

there is no valid assignment. If the nodes in region A are all successfully assigned the direction *Left*, then go to the bottom row in the region B and repeat the procedure described until now.

The above algorithm visits every node exactly once; hence, it is of linear complexity. The correctness of the above algorithm can be proved by justifying each step of the algorithm. □

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. 4(b), where the permitted directions are *Left*, *Right*, and *Top*. An algorithm for finding a valid assignment can be described as follows.

Sequentially examine the rows of the grid starting with the bottom row of the grid. Depending on the number of nodes belonging to F in the row make the following assignments.

1) If the row has only one node belonging to F, then assign it the direction *Left* or *Right* and move to the next upper row.

2) If the row has two nodes, then assign the left node the direction *Left* and the right node the direction *Right*; move to the next upper row.

3) If the row has three or more nodes, then assign the leftmost direction *Left*, the rightmost node the direction *Right*, and the nodes in the middle the direction *Up*. The rest of the grid now gets partitioned into several distinct regions as shown in Fig. 4(b). The nodes in the inner regions have only one permissible direction and hence they are assigned the direction *Up*. If such assignments lead to contradiction (i.e., there are two nodes in the same column in one of these regions), then there is no valid assignment.

Now, in each of the two outer regions, namely A and B, there are two permissible directions that are at right angles. Hence, each such region can be tested for valid assignments by following the algorithm outlined in Lemma 3.

The above algorithm visits every node only once and hence it is of linear complexity. Also, note that the above algorithm is a combination of the algorithms developed in Lemma 2 and Lemma 3. In particular, the algorithm follows the assignment procedure in Lemma 2 until it finds a row with three or more nodes of F, and from then on it follows the algorithm in Lemma 3. A formal proof of correctness for the above algorithm will be skipped here; however, we should note here that a justification for the above algorithm can be constructed by combining the justifications for Cases 1 and 2. □

We should note here that the algorithms outlined in Lemmas 2, 3, and 4 (and the ones to be presented later in this section)

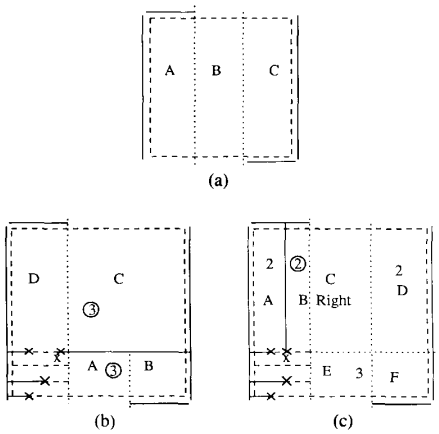


Fig. 5. The special case corresponding to Case 4 in Section II-A.

can be easily adapted to work for *restricted versions* of the respective cases. For example, in Case 3, one can restrict possible assignments to a particular node in F by disallowing the assignment of one of the feasible directions to it. A possible motivation behind such a restriction may be to indicate that the spare processor along the forbidden direction is itself faulty, and hence one cannot allow a compensation path in that direction.

There are several ways of adapting the above algorithms; however, one particularly simple one is described here. Without loss of generality, assuming that the direction *Left* is disallowed for the node $x \in F$; this restriction is meaningful only if there is no node to the left of x in the same row as x. In order to accommodate this restriction, augment the set F with a node y and place it in the same row as x, but on the extreme left edge of the grid in a newly added column. Now in the augmented problem, the direction *Left* is no longer feasible for x; moreover, since the node y is at the very edge of the grid, it can be always assigned the direction *Left* without affecting possible assignments to the rest of the nodes in F (see Lemma 1).

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. 5(a). 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. The motivation is the same as before: to identify nodes that can be assigned directions which do not interfere with the rest.

If the row under consideration has only one node in the region A, then assign it 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 two or more 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. 5(b) 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 2.

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 have already discussed how to adapt algorithms to accommodate such restrictions; 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. 5(b)].

2) If the result of the previous search is in negative, then assign x the direction *Up*; Fig. 5(c) 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. 5(c)].

Note that if none of the rows in the region A of Fig. 5(a) has more than one node 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. 5(a), 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. □

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.

Part 1: In the first part of the algorithm, one attempts to determine valid directions using the principle of Lemma 1, 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. If there is a node that cannot be assigned the direction *Left*, then go to the next step.

2) Sequentially examine the rows of the reduced grid (i.e., the portion of the original grid that is unassigned in Step 1) starting with the top row. If possible, then assign to every

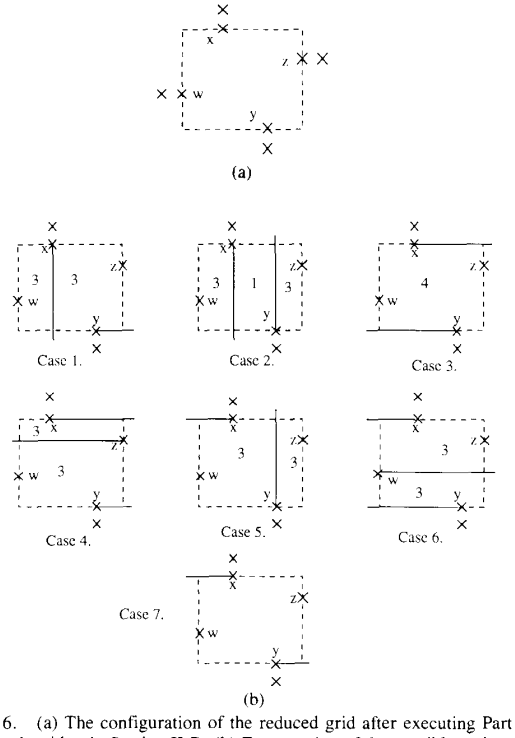


Fig. 6. (a) The configuration of the reduced grid after executing Part 1 of the algorithm in Section II-B. (b) Enumeration of the possible assignments of directions to nodes x and y.

node in the row under consideration the direction *Up*. If all nodes can be assigned the direction *Up*, then go to the next row, else go to the next step.

3) Sequentially examine the columns of the reduced grid (i.e., the portion of the original grid that is unassigned after Steps 1 and 2) starting with the rightmost column. If possible, then assign to every node on the column the direction *Right*. If all nodes can be assigned the direction *Right*, then go to the next row, else go to the next step.

4) Sequentially examine the rows of the reduced grid (i.e., the portion of the original grid that is unassigned after Steps 1, 2, and 3) starting with the bottom. If possible, then assign to every node on the row the direction *Down*. If all nodes can be assigned the direction *Down*, then go to the next row, else go to the next step.

5) After completion of the four steps, one has a reduced grid which has been obtained by peeling off the outer layers of the original grid. Next, repeat Steps 1–4, until the reduced grid is of the form shown in Fig. 6(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. 6(a). The objective is to show that we can determine such valid directions in linear time.

Let us assume, without loss of generality, that each of the outermost rows and columns in the grid, shown in Fig. 6(a), contains only one node of F; if an outer row or column contains more than one node, then the search for valid directions is going to be simpler and we shall mention such simplifications later in this section.

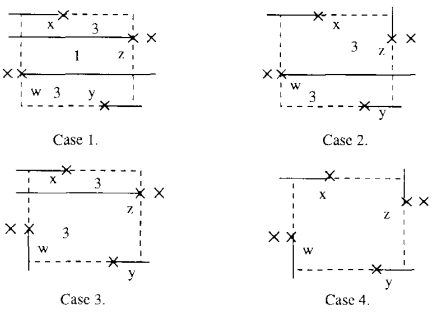


Fig. 7. Enumerating the possible assignments of directions to nodes w and z in Case 7 of Fig. 6.

Let us consider the nodes x and y in Fig. 6(a); each node has at most three possible directions that can be assigned to it. It turns out that altogether there are seven possible ways in which x and y can be assigned directions [see Fig. 6(b)], and one has to check for valid assignments under each of the assignments. However, each of the first six cases can be searched for valid assignments by breaking up the regions into special cases (or their restricted versions) that were considered in Section II-A. The breaking up of the regions and the special cases they correspond to are shown in Fig. 6(b). Note that the algorithms developed for the special cases are of linear complexity; hence, each of the first six cases can be searched for valid assignments in linear time.

The seventh case, where the node x is assigned *Left* and the node y is assigned *Right*, can be further simplified by considering possible assignments to nodes w and z. Each of the nodes w and z has only two permissible directions and they together lead to four cases (see Fig. 7). Three of the cases can be searched for valid assignments in linear time by using the algorithms developed for the special cases.

Thus, a complete algorithm for a systematic search of a valid assignment for the nodes in the outer layers of the grid shown in Fig. 6(a) can be summarized as follows.

1) Use the algorithms developed for special cases to search for valid assignments in each of the first six configurations, which are illustrated in Fig. 6(b). 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.

2) Enumerate the seventh case as shown in Fig. 7, and search for valid assignments for the first three enumerations.

3) If every search in Steps 1 and 2 fails then assign directions to nodes x, y, w, and z that are shown in Case 4 of Fig. 7, 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 Steps 1 and 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 4 in Fig. 7) is the only candidate.

We can make the following remarks about the procedure discussed so far.

1) Until now we have assumed that every outermost row or column in the reduced grid shown in Fig. 6(a) has only one node. Consider, however, the case where in addition to x, there are other nodes in the upper outermost row of the grid. This would imply that the possible directions that could be assigned to x are restricted and some of the cases shown in Fig. 6(b) are not valid. Hence, the search for valid assignments becomes simpler. The same comment holds if all the four nodes (i.e., x, y, w and z) are not distinct.

2) The relative positioning of the nodes w and z in Fig. 6(a) has also been chosen to illustrate the worst case situation. Consider, for example, the case where the node z is located lower than w. Then, one can easily show that even Case 4 of Fig. 7 can be searched for valid assignments by utilizing the algorithm for the Special Case 4, that was discussed in Section II-A.

Theorem 1: There is a quadratic time algorithm ($O(|F|^2)$) 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 $|F|$ such layers in the grid, the complexity of the algorithm is $O(|F|^2)$. □

C. Avoiding Near-Miss Situations

We can make the following remarks.

1) A near-miss situation can occur only among nodes in F that are in adjacent rows and columns. Hence, if there is more than one consecutive row (column) that contain none of the vertices in F, then clearly all but one of these blank rows (columns) can be deleted. Thus, without loss of generality one can assume that $1 \leq m, n \leq 2|F|$.

2) While discussing the algorithms without considering near-miss situations, only cases that needed new algorithms are Cases 1, 2, and 3 (see Section II-A). The algorithms for the general case as well as Case 4 were developed by judiciously making use of the algorithms for the other special cases. The same argument holds when one considers near-miss situations; i.e., one needs to develop algorithms only for the three special cases. However, Case 2 can never have a near-miss situation; this is because the two permissible directions are always at right angles and can never be running opposite to each other. Thus, we have developed algorithms only for Cases 1 (line segments along two opposite directions are allowed) and 3 (line segments along at most three directions are allowed).

Lemma 6: There is a linear time algorithm for determining a valid assignment for Case 1 even when near-miss situations are avoided.

Proof: Let us assume that the directions permitted are *Left* and *Right*. Assume also that none of the rows has more than three nodes from F; if any row does, then obviously there is no valid assignment(s) for the node(s) in the middle. Also, note that because of near-miss situations, the rows are no longer independent and assigning directions to the nodes in one row may affect possible assignments of directions to nodes in adjacent rows. The basic idea behind the algorithm is: since in the rows with two nodes the directions are fixed, one can systematically assign directions to the single nodes such that near-miss situation does not occur. The algorithm can be briefly described as follows (for more details see [13]): 1) Sequentially examine the rows from bottom until a row with two nodes is reached. 2) Now suppose there is a row with two nodes of F, then the node on the left side is assigned the direction *Left* and the node on the right is assigned the direction *Right*. Since these directions are fixed, one can go down and assign appropriate directions to the single nodes in the lower rows. Stop when either a) a row is encountered for which the direction is not determined by the assignment to the node in the row below it; in this case, the rows below do not interact with the rows on top and the whole algorithm can be repeated again, or b) another row with two nodes is encountered and there is no valid assignment. 3) If the procedure has not failed and all the rows have not been examined, then repeat Steps 1 and 2 for the unassigned rows of the grid. □

Lemma 7: There is a linear time algorithm for determining a valid assignment for Case 3 even when near-miss situations are avoided.

Proof: Let us assume that the permitted directions are *Left*, *Right*, and *Up*. We can divide the rows into two regions, i.e., examine the rows from the bottom until a row with more than two nodes (belonging to F) is found; let the region above and including this row be called B, and the region below be called A. We know from Lemma 4 how to handle the region B, i.e., it gets partitioned into two instances of Case 2. One, however, has to make assignments in the region A, such that there are no near-misses; it can be done by applying the algorithm developed in Lemma 6. For more details see [13]. □

We should mention here that efficient data structures can be easily designed to implement all the algorithms described in this section in the claimed time complexities.[1] One way of implementing the algorithms would be to bucket-sort the points in F according to their row and column numbers.

III. Efficient Algorithms for Reconfiguration in More General Models

Fig. 3 shows an augmented single-track model: it has more switches compared to the simple single-track model, and each switch has one more state than before. We should note here that by adding the fifth state, the complexity of and the area taken by a switch is not appreciably increased; moreover, it is a common practice to assume that a switch in the single track model will have the five states shown in Fig. 3 [12].

One would expect that the added hardware (i.e., more switches) in the new model should facilitate reconfiguration. It is indeed the case, and Fig. 8 shows a valid reconfiguration when one introduces a bend in the compensation path; the modified routing only affects the states of the switches that are local to the compensation paths. As mentioned before, it is not possible to have bent compensation paths in the simple single-track model [1], [3]. This added flexibility gives more

[1] In [14], an improved $O(|F| \log |F|)$ algorithm has been presented for Problem 1.

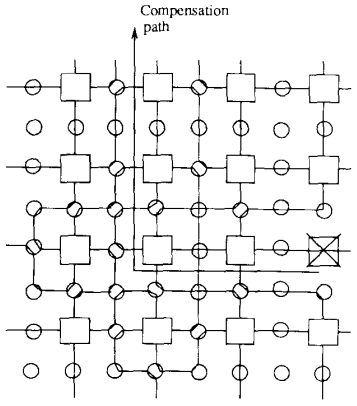


Fig. 8. A bent compensation path and a corresponding routing scheme in the augmented single-track model.

power, e.g., in the example shown in Fig. 10, the faulty processor x is no longer blocked and can be replaced easily in the augmented model by a bent compensation path. The augmented model, however, is still not powerful enough to handle near-miss situations and possible complications due to adjacent, but bent compensation paths.

For multiple-track models, however, it is sufficient for the compensation paths to be continuous and nonintersecting. Hence, a generalized sufficient condition for the multiple-track models can be stated as follows: an array grid model with multiple-track switches is reconfigurable if one can determine a set of nonintersecting and continuous compensation paths (not necessarily straight) for the faulty PE's in the array. Now checking whether for an array with a given distribution of faulty processors one can determine such compensation paths, is equivalent to solving the following combinatorial problem:

Problem 2: Let V be the set of grid points in an $n \times n$ two-dimensional rectangular grid, and let $F \subset V$. Determine a set of nonintersecting paths in the grid such that

a) each vertex $v \in F$ is connected to a distinct node on the boundary of the grid by one of the paths.

b) the paths are nonintersecting.

Theorem 2: There exists a polynomial (in n) time algorithm for solving Problem 2.

Proof: We shall prove the theorem by reducing Problem 2 to the well-known MAX-FLOW problem for which there are several efficient polynomial time algorithms [7]. The desired network for the flow problem can be described by a directed graph $G = (V', E)$ where

1) $V' = V \cup \{s, t\}$, where V is the set of nodes in the grid, s is the source node, and t is the sink node.

2) E consists of three types of arcs: 1) for every pair of nodes i, j that are adjacent in the grid, two arcs $i \rightarrow j$ and $j \rightarrow i$ are in E, 2) for every boundary node $v \in V$, an arc connecting it to the sink node t (i.e., the arc $v \rightarrow t$) is in E, and 3) for every node $v \in F$, an arc connecting the source node s to it (i.e., the arc $s \rightarrow v$) is in E.

3) The capacity of every $e \in E$, denoted by $c(e)$, is unity.

4) The capacity of every node $v \in V$ (i.e., every node in V', except the source node s, and the sink node t), denoted by $c(v)$, is unity.

Fig. 9 shows a grid and the corresponding network derived

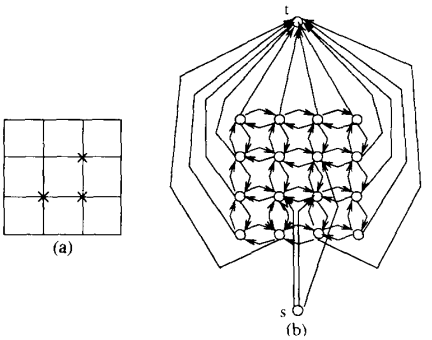


Fig. 9. Construction of the network G in Theorem 2: (a) shows a grid with points belonging to F (marked by cross signs); (b) shows the corresponding network G.

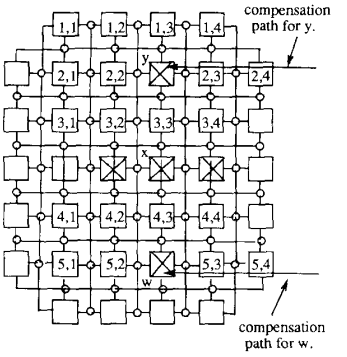


Fig. 10. Example of valid reconfiguration in a case where the sufficient conditions in the reconfigurability theorem are not satisfied.

according to the above rules. We can make the following observations about the network derived above.

1) Since the out-degree of the source node is $|F|$ and every edge has unit capacity, the maximum possible flow in the network is $|F|$.

2) Every unit of flow pushed from the source to the sink defines a unique directed path connecting a node in F to one of the boundary nodes. This follows directly from the following facts: a) all edges have unit capacity b) the only way the source node can push a unit of flow is through one of the nodes in F, and c) the only way the sink node can receive a unit of flow is through one of the boundary nodes.

3) Since every node (except the source and the sink nodes) in the network has unit capacity, only one edge coming into a node can be saturated, i.e., has unit flow. Hence, every unit of flow in the network not only defines a unique path, but the corresponding paths are also nonintersecting.

4) A path defined by a unit flow in the directed network is also a path in the undirected grid (excluding of course the edges from the source and to the sink). This follows from the fact that if the arc $i \rightarrow j$ in G is saturated, then the reverse arc $j \rightarrow i$ cannot be saturated; if it is, then the two arcs will form a circular flow which does not contribute to the net flow from the source to the sink.

Our claim is that, *there is a solution to Problem 2 if and only if the maximum flow in the network G*, as defined above, is $|F|$. It follows directly from the above remarks that if the maximum flow in the network G is $|F|$, then there are

$|F|$ nonintersecting paths in the corresponding grid such that each one connects one of the nodes in F to a node in the boundary. Each such path is defined by a unit flow from s to t in the network G.

Now if there is a solution to Problem 2, then we know that there is a set P, of $|F|$ nonintersecting paths in the grid connecting each node in F to a distinct boundary point in the grid. One can make the paths in P directed in the corresponding network G by choosing the appropriate arcs such that the paths are from the nodes in F to the respective boundary points. One can now determine a flow $|F|$ in the network G as follows: 1) assign unit flow to all the arcs coming out from the source node s, and the arcs lying on the paths in P, 2) assign unit flow to the arcs that join the boundary points belonging to the paths in P, to a sink node, and 3) assign 0 flow to the rest of the arcs in E. By construction then, the network has flow $|F|$. □

Note that by solving the corresponding flow problem, one not only determines whether Problem 2 is feasible, but one also gets a required set of nonintersecting paths when it is possible to do so. Also, since in the network all edges and nodes have unit capacities, the complexity of the maximum-flow algorithm can be shown to be $O(n^3)$ [7].

IV. Concluding Remarks

In this paper, we have addressed some of the combinatorial problems that arise while reconfiguring processor arrays with fixed hardware resources. In the case of an array grid model with single-track switches, the problem of reconfiguring the array can be reduced to the combinatorial problem of determining a set of nonintersecting straight lines in a grid. In the case of more general models such as the array grid models with multiple-track switches, the reconfigurability problems can be reduced to the problem of determining a set of nonintersecting paths in a two-dimensional grid. For the first problem we provide an efficient geometric algorithm while for the latter problem we derive an efficient algorithm by reducing it to a MAX-FLOW problem. A few remarks.

1) *Failures in Switches/Connections:* In this paper we have considered only processor failures. However, often switch failures are modeled by disallowing certain compensation paths or by considering that certain PE's are faulty (see [3]). We have mentioned in Section II how our algorithm can be adapted if, for individual faulty processors, compensation paths in certain directions are not allowed. In that sense, *we have shown that reconfiguration in single-track models, even with switch failures, is of polynomial complexity.*

Moreover, another reason for concentrating on processor failures is that the yield for switches and connections is much higher than the corresponding yield for processors [4].

2) *Hexagonal/Triangular Grid:* The general topology considered in this paper is that of a rectangular grid; however, one would be interested in addressing similar issues for other kinds of grids. A little thought will show that Problem 2, when translated to any other grid, can still be solved in polynomial time by using MAX-FLOW algorithms. It is not clear, however, how to solve Problem 1 for grids with arbitrary topologies.

Appendix
The Sufficient Conditions for Reconfigurability are not Necessary

Herein we shall show why the conditions in the reconfigurability theorem [1], [3] are not necessary. The reason lies in the fact that in the theorem nothing is said about compensation paths for healthy processors; however, as we are going to illustrate, one can gain by replacing healthy processors too. Consider for example, the faulty processor x in Fig. 10; it is blocked by faulty processors on all four sides and according to the reconfigurability theorem one cannot determine a valid compensation path for x. However, as shown in Fig. 10, one can still reconfigure the given array by replacing the whole row containing the processor x by one of the spare rows; the faulty processors that are left (i.e., y and w) can be replaced by the usual compensation paths. For more examples see [13].

References

[1] S. N. Jean and S. Y. Kung, "Necessary and sufficient conditions for reconfigurability in single-track switch WSI arrays," in *Proc. Int. Conf. Wafer Scale Integration*, Jan. 1989.

[2] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[3] 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.

[4] T. Leighton and C. E. Leiserson, "Wafer-scale integration of systolic arrays," *IEEE Trans. Comput.*, vol. C-34, no. 5, pp. 448–461, May 1985.

[5] F. Lombardi, M. G. Sami, and R. Stefanelli, "Reconfiguration of VLSI arrays by covering," *IEEE Trans. Comput.-Aided Design*, 1989.

[6] W. R. Moore, "A review of fault-tolerant techniques for the enhancement of integrated circuit yield," *Proc. IEEE*, pp. 684–698, May 1986.

[7] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[8] A. L. Rosenberg, "The Diogenes approach to testable fault-tolerant array of processors," *IEEE Trans. Comput.*, pp. 902–910, Oct. 1983.

[9] M. Sami and R. Stefanelli, "Reconfigurable architectures for VLSI processing arrays," *Proc. IEEE*, pp. 712–722, May 1986.

[10] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*. Bedford, MA: Digital, 1982.

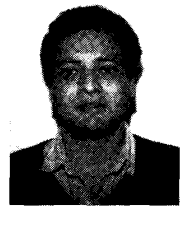
[11] L. Snyder, "Introduction to the configurable, highly parallel computer," *IEEE Trans. Comput.*, vol. C-15, pp. 47–56, Jan. 1982.

[12] S. K. Tewksbury, *Wafer-Level Integrated Systems: Implementation Issues*. New York: Kluwer Academic, 1989.

[13] V. Roychowdhury, J. Bruck, and T. Kailath, "Efficient algorithms for reconfiguration in VLSI/WSI arrays," Tech. Rep., Stanfor Univ., Nov. 1989.

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

[15] J. W. Greene and A. El Gamal, "Configuration of VLSI arrays in the presence of defects," *J. ACM*, vol. 31, no. 4, pp. 694–717, Oct. 1984.



Vwani P. Roychowdhury was born in Asansol, India, on April 16, 1961. He received the B. Tech degree from the Indian Institute of Technology, Kanpur, the M.S. degree from University of Rochester, Rochester, NY, and the Ph.D. degree from Stanford University, Stanford, CA, in 1982, 1983, and 1988, respectively, all in electrical engineering.

He is currently associated with the Information Systems Laboratory at Stanford University as a Research Associate. His research interests include parallel algorithms and architectures, special purpose

computing arrays and VLSI design, fault-tolerant design, and the theory of neural networks.

Jehoshua Bruck (M'90) was born in Haifa, Israel, on April 19, 1956. He received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion-Israel Institute of Technology, Haifa, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1989.

From 1982 to 1985 he was with the IBM Haifa Scientific Center, Israel. In March 1989, he joined the IBM Research Division at the Almaden Research Center, San Jose, CA, where he is presently a research staff member. His research interests include algorithms, computational complexity, error-correcting codes, fault-tolerant computing, parallel computing, and neural networks.

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).

After a year at the Jet Propulsion Laboratories, Pasadena, CA, he joined Stanford University, Stanford, CA, as an Associate Professor of Electrical Engineering in 1963. He was promoted to Full Processor in 1968, served as Director of the Information Systems Laboratory from 1971-1980, as Associate Department Chairman from 1981 to 1987, and currently holds the Hitachi America Professorship in Engineering. He has worked in a number of areas including information and communication theory, signal processing, linear systems, linear algebra, operator theory, and control theory. His recent research interests include array processing, fast algorithms for nonstationary signal processing, and the design of special purpose computing arrays. 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 and Churchill fellowships, among others, and received awards from the IEEE Information Theory Society, the IEEE Signal Processing Society and the American Control Council. He served as President of the IEEE Information Theory Group in 1975. He is a Fellow of the Institute of Mathematical Statistics and is a member of the National Academy of Engineering.