Journal of VLSI Signal Processing, 3, 329–344 (1991)
© 1991 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

# New Algorithms for Reconfiguring VLSI/WSI Arrays

THEODORA A. VARVARIGOU, VWANI P. ROYCHOWDHURY AND THOMAS KAILATH*
*Information Systems Laboratory, Stanford University, CA 94305*

Received July 12, 1990; Revised April 15, 1991.

**Abstract.** In this paper we present new algorithms for reconfiguring arrays of identical Processing Elements (PEs) in the presence of faults. In particular, we consider a well-studied reconfiguration model which consists of a rectangular array of PEs with spare columns of PEs on one side. In the presence of faulty PEs, reconfiguration is achieved by constructing a *logical array* using only the healthy non-spare and spare PEs. Note that one can always successfully reconfigure the array as long as the number of faulty PEs is no more than the number of spare PEs. The general objective, however, is to derive a logical array such that the geometric distances between *logical neighbors* (i.e., PEs that are connected in the reconfigured array) are kept small. This criterion is motivated by the fact that shorter interconnects reduce the communication delays among the PEs, and also lead to less routing hardware. The problem of determining a reconfiguration that minimizes the length of the longest interconnect is *hard* and several researchers have presented sub-optimal algorithms that seem to have satisfactory performance. In this paper we develop a *new efficient algorithm* that can reconfigure any array with arbitrary patterns of faulty PEs. Furthermore we show that our algorithm performs better than most of the other algorithms developed for similar models.

## 1. Introduction

Fault tolerance has been extensively studied in the recent literature as a way of improving reliability and production yield in VLSI devices. Fault tolerant techniques have already been implemented in many applications i.e., memory cells [1] etc., and have improved their performance considerably. Fault tolerance issues have also been addressed for array processor architectures (see e.g., [2]–[10]) because of

1. their wide application in signal processing, matrix multiplication, inversion, etc.,
2. their low manufacturing cost, and
3. the difficulty of entirely fault-free implementation of large area circuits.

Fault tolerance can be incorporated at different levels of the design-hierarchy, e.g., (1) the *cell level*, which addresses questions on the detection of faults in the smallest replaceable element of the array; self-checking processors have been proposed for this level of fault detection as well as system level techniques; (2) the *routing level*, which considers failures in the routing hardware of the array; and (3) the *array level*, which addresses questions about how to replace the fault cells with spares that are placed in the array in a predetermined fashion in order to keep the need for additional routing hardware as small as possible.

A considerable amount of research has been carried out to incorporate fault tolerance at the *array level* [5], [6], and a number of different reconfigurable models for rectangular arrays of identical PEs have been proposed. For example, Sami and Steffanelli [8] have presented a multiplexer-based redundant interconnection scheme that results in a significant improvement of the array yield. Other reconfiguration models that use multiple-track routing channels and programmable switches have been studied extensively in [4], [7], [9], [10].

In this paper we consider the model studied in [2], [8]: it consists of an $n \times k$ rectangular array with $m$ columns of spare PEs on one side. The routing hardware used for reconfiguration could vary: in [8] a multiplexer based interconnection scheme is used whereas in [2] the routing is implemented by tracks and switches in a similar way to the routing hardware of the models presented in [3], [4], [7], [9].

In the presence of faulty PEs, reconfiguration is achieved by constructing a *logical array* using only the healthy non-spare and spare PEs. Note that one can always successfully reconfigure the array as long as the

*This work was supported in part by the SDIO/IST U.S. Army Research Office through Contract DAAL03-90-G-0108.

number of faulty PEs is no more than the number of spare PEs; an arbitrary construction of the logical array, however, might result in large interconnect lengths. *The general objective for reconfiguration in such models is to minimize the geometric distances between logical neighbors* (i.e., PEs that are connected in the reconfigured array). This criterion is motivated by the fact that shorter interconnects reduce the communication delays among the PEs, and also might lead to less routing hardware. The manner in which the size of the additional routing hardware depends on the interconnect lengths has been studied for the multiplexer based scheme in [8] and also for the tracks-switches based scheme in [2].

The problem of determining a reconfiguration that minimizes the length of the longest interconnect is *hard* and several researchers [2], [8] have presented *suboptimal algorithms* that seem to have satisfactory performance. It is not clear, however, that the algorithms presented in the literature make full use of the capability of the above model, e.g., the algorithms presented in [8] fail to reconfigure even when the faulty PE distributions are very simple (see Section 4 for examples). In this paper we develop a *new efficient algorithm* that is simple and at the same time can reconfigure any array, with arbitrary fault distribution, using only small interconnect lengths. We can also demonstrate that our algorithm performs better than other algorithms that use the same model. In particular, we prove that our algorithm performs better than the one presented in [8]: It can reconfigure (1) *all* the faulty patterns in [8], *using the same or smaller interconnect lengths*, and (2) much more general faulty patterns, again using the same interconnect lengths permitted in [8]. As a comparison to the algorithm in [2], we show that our algorithm can reconfigure certain faulty patterns while maintaining the length of links between the PEs constant, whereas the algorithm presented in [2] needs length of links that grow proportionally to the size of the array.

The key idea that allows our algorithm to perform better than others that use the same model is the *local treatment of the faulty patterns*. The local treatment of the faulty pattern is achieved by segmenting the given array into subarrays of special structure that are easy to reconfigure. These special faulty patterns are then reconfigured in a way that keeps the interconnection length requirements small; this leads to a reconfiguration of the whole array with small interconnect lengths.

An overview of the rest of the paper is as follows. In Section 2 we present a reconfiguration algorithm for the case of only one column of spare PEs along one side of the array. The algorithms are based on one simple algorithm that shows how to reconfigure an $N \times (N + 1)$ array into an $(N + 1) \times N$ array and vice versa. In Section 3 we shall generalize the algorithm of Section 2 to the case where there are more than one spare columns of PEs. Section 4 presents some discussion on the evaluation of our algorithm and compares it to other reconfiguration algorithms that use the same or similar types of models. Section 5 presents some concluding remarks. Finally, in the Appendix we discuss time complexity issues, of the algorithms developed in this paper.

## 2. Reconfiguration Algorithms

In this section we present a new algorithm for reconfiguring processor arrays according to the model already described in the Introduction. For the sake of simplicity in illustrating the algorithms, we shall first consider the case where there is only one column of spare PEs in one side of the array. The reconfiguration for the case where there are multiple spare columns is similar and will be outlined in the next section.

In the first part of this section we shall introduce a novel way of reconfiguring an $N \times (N + 1)$ array into an $(N + 1) \times N$ array. This is a simple reconfiguration procedure that will be later used to develop reconfiguration algorithms for certain special cases of faulty patterns. Reconfiguration algorithms for the general case of arbitrary faulty patterns will then be constructed by utilizing the algorithms for the special cases. Before proceeding to the analysis of the reconfiguration procedure, let us define the following:

*Definition 1.* Physical array (column, row) is the given $N \times (N + 1)$ array (column, row).

*Definition 2.* Logical array (column, row) is the array (column, row) which we derive after the reconfiguration procedure.

### *2.1. Basic Reconfiguration Algorithm: Reconfiguring an $N \times (N + 1)$ Array into an $(N + 1) \times N$ Array*

Herein, we develop a procedure for reconfiguring an array that has one more column than rows into an array that has one more row than columns. The idea behind our algorithm is to increase the length of each of the first $N$ columns of the array by using the PEs

of the $(N + 1)^{th}$ column. The way of doing such a reconfiguration can be described as follows: we construct the first logical column of the logical (reconfigured) array using two PEs from the first physical row of the physical array and one PE from each of the following physical rows. In general we construct the $i^{th}$ logical column of the logical (reconfigured) array using two PEs from the $i^{th}$ physical row and one PE from the rest of the physical rows as shown in figure 1(a). We formally describe this mapping of the entries of the logical array into the entries of the physical array in a code form as follows:

Consider entry $(i, j)$ of the physical array, where $1 \leq i \leq N$ and $1 \leq j \leq N + 1$. Map the $(i', j')$ entry of the logical array to the entries of the physical array in the following way:

$$(i', j') \rightarrow \begin{cases} (i, j) & \text{if } i' = j' \\ (i - 1, j) & \text{if } i' - j' \geq 2 \\ (i, j + 1) & \text{if } j' - i' \geq 1 \\ (i - 1, j + 1) & \text{if } i' = j' + 1 \end{cases}$$

The procedure for mapping a $(N + 1) \times N$ physical array into an $N \times (N + 1)$ logical one is similar. We simply use the procedure above, interchanging the rows with the columns and vice versa.

### 2.2. *Reconfiguration Algorithms for Special Cases*

We now describe efficient reconfiguration algorithms for the following Special Cases (we shall use these reconfiguration algorithms later for the general case):

*Special Case 1.* An $N \times (N + 1)$ physical array, with $N$ faulty PEs, each in a different physical column, into an $N \times N$ logical array of healthy PEs.

*Special Case 2.* An $N \times N$ physical array that has only one faulty processor in each row, into an $(N - 1) \times N$ logical array of healthy PEs.

*Special Case 3.* An $K \times (N + 1)$ physical array into a logical array that has $N$ logical columns, out of which, $L$ predetermined ones, namely $c_i$ $i = 1, \ldots, L$ are of length $(K + k_i)$ $i = 1, \ldots, L$, and the rest $(N - L)$ columns are of length $K$, where $\Sigma_{i=1}^{L} k_i = K$.

Before proceeding to the description of the algorithm for Special Case 1, we shall define the following:

*Definition 3.* An $i$-fault physical (logical) column (row) $i = 0, 1, 2$ is a physical (logical) column (row) that has $i$ faulty PEs.

*Special Case 1.*

In this reconfiguration algorithm we shall reconfigure an $N \times (N + 1)$ array with $N$ faulty PEs, at most one in each column, into an $N \times N$ healthy one. The reconfiguration procedure can be outlined as follows:

1. We first apply the Basic Reconfiguration Algorithm described in Section 2.1 as if there were no faulty PEs in the array (see figure 1(c)). The resulting logical array has logical columns with 0, 1 or 2 faulty PEs. This is because every logical column has PEs from only two physical columns, as is apparent from the Basic Reconfiguration Algorithm. However, in this Special Case every physical column has at most one faulty PE; therefore, the maximum number of faulty PEs that any logical column can have is two.
2. Now, the desired reconfigured array should have $N$ PEs in each column. Hence, the 2-fault logical columns (in the logical array obtained after the above step) that have $(N - 1)$ healthy PEs, need to borrow one PE each from the 0-fault columns that have $(N + 1)$ healthy PEs. The 1-fault columns have $N$ healthy PEs and need not change the number of their PEs.

   It is easy to see that for every 2-fault logical column that needs an extra PE, there is a corresponding 0-fault logical column that has an available PE to give. For each such pair of logical columns we can define a *borrowing process* such that a 2-fault logical column *borrows* the extra PE of the corresponding 0-fault logical column. This borrowing process between the 2-fault columns and the 0-fault columns takes place along the $N^{th}$ physical row of the array as shown in figure 1(d). Lemma 6 in Section 4.1 shows that these borrowing processes do not interfere with each other, i.e., no more than one such borrowing process can occur in the sample columns of the $N^{th}$ physical row.

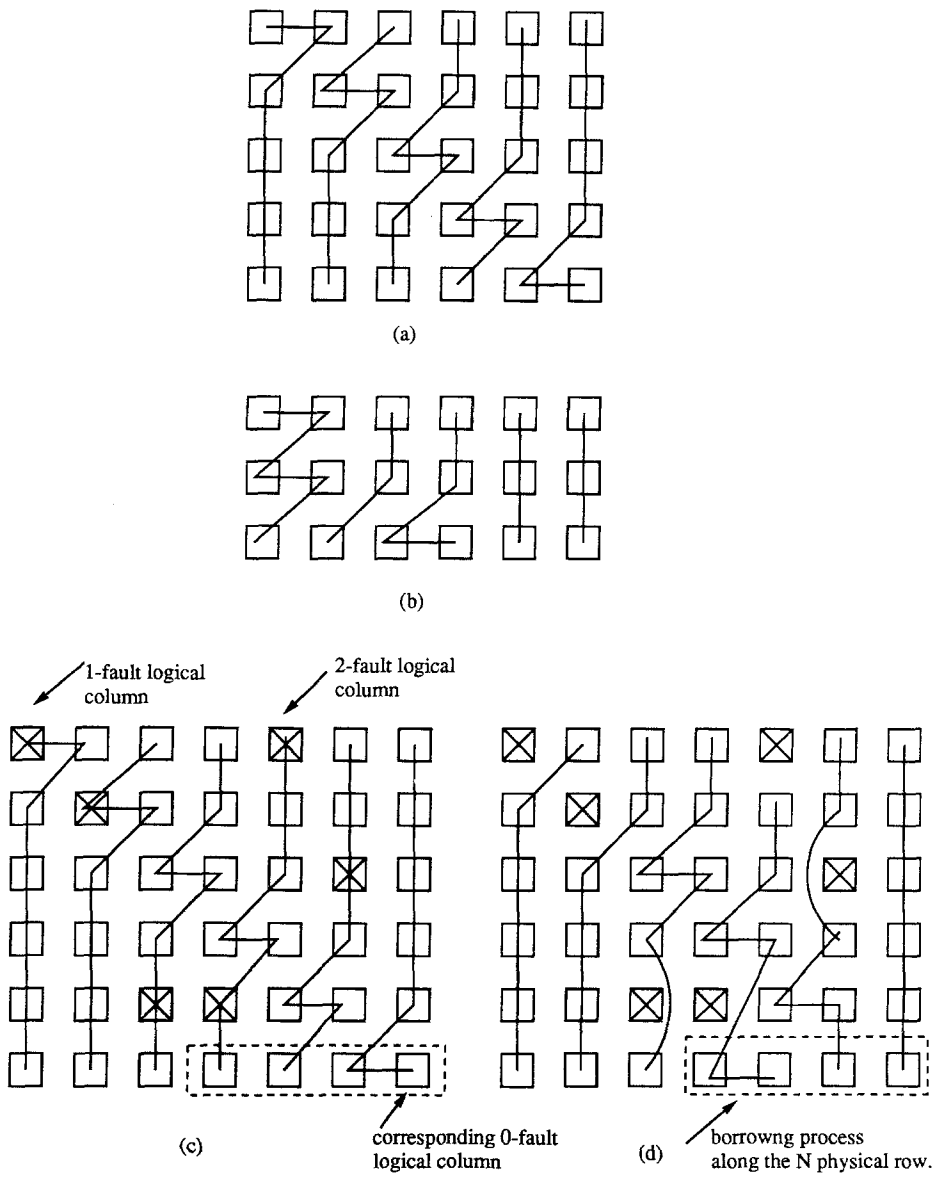



*Fig. 1.* Reconfiguration for Special Cases.

The reconfiguration procedure presented above can be formally described in a coded form as follows:

1. Construct an $(N + 1) \times N$ logical array out of the $N \times (N + 1)$ physical (given) one according to the basic reconfiguration algorithm of Section 2.1.
2. For the logical columns $j'$ with one faulty processor in the entry $(k'\, j')$, do the following renaming:

$$(i', j') \rightarrow \begin{cases} (i', j') & \text{if } i' < k' \\ (i' - 1, j') & \text{if } i' > k' \end{cases}$$

3. For the 2-fault logical columns $j'$ with faults in entrees: $(k'\, j')$ and $(l'\, j')$, $(l > k)$, do the following renaming:

$$(i', j') \rightarrow \begin{cases} (i', j') & \text{if } i' < k' \\ (i' - 1, j') & \text{if } k' < i' < l' \\ (i' - 2, j') & \text{if } l' < i' \end{cases}$$

4. For the 2-fault columns $j'$, if $l'$ is the corresponding 0-fault column, do the following renaming:

$$(N, k') \rightarrow \begin{cases} (N, k' + 1) & \text{if } j' < k' < l' \\ (N + 1, l') & \text{if } k' > l' - 1 \end{cases}$$

*Special Case 2.*

We now present our algorithm for reconfiguring an $N \times N$ array with $N$ faulty PEs, each in a different physical row, into an $(N - 1) \times N$ logical healthy array. The reconfiguration procedure for this Special Case can be outlined as follows:

1. Consider the $N \times N$ array shown in figure 2(a). Add one auxiliary fault-free row on the top of the physical array as shown in figure 2(a). The resulting array is an $(N + 1) \times N$ array that has $N$ aulty PEs, one in each row.
2. Apply the reconfiguration algorithm for Special Case 1 presented above (interchanging the role of the columns and rows) to get a $N \times N$ healthy array (see figure 2(b)).
3. Disregard the first row of the reconfigured array. This results in a $(N - 1) \times N$ healthy array achieving the goal of this reconfiguration procedure (see figure 2(b)).

*Special Case 3.*

We now describe our algorithm for reconfiguring an $K \times (N + 1)$ array into an array of $N$ columns such that (1) $L$ special columns, namely $c_1, \ldots, c_L$, are of length $(K + k_i)$, $i = 1, \ldots, L$, with $\Sigma_{i=1}^{L} k_i = K$, and (2) the rest of the $(N - L)$ columns are of length $K$ each. We shall refer to the columns $c_1, \ldots, c_L$ as the *special* columns of the array and the corresponding $k_i$ will be defined as the *degree* of the *special* column $c_i$. For example, the $3 \times 6$ array of figure 1(b) is reconfigured into an array that has its first and third columns extended by two and one PEs respectively. The *special* columns of the array are: $c_1 = 1$ and $c_2 = 3$, and the corresponding *degrees* are: $k_1 = 2$ and $k_2 = 1$. (We realize that the usual definition of the *array* implies that all the columns have the same number of entries. For the sake of simplicity, we shall refer to the block structures that have some of their columns extended as arrays as well.)

The idea for this reconfiguration procedure is similar to that for reconfiguration of an $N \times (N + 1)$ physical array into an $(N + 1) \times N$ logical one presented in Section 2.1. We simply use the $K$ PEs of the $(N + 1)^{\text{th}}$ physical column of the array to increase the length of the predetermined columns $c_1, \ldots, c_L$. The methodology for this kind of reconfiguration can be described as follows:

1. For logical column $c_1$ use two PEs from the first $k_1$ physical rows and one PE from every other physical row. For logical column $c_i$ use two PEs from the $\Sigma_{l=1}^{i-1} k_l + 1, \ldots, \Sigma_{l=1}^{i} k_l$ physical rows and one PE from every other physical row.
2. For any other logical column except $c_i$'s, use one PE from each physical row.

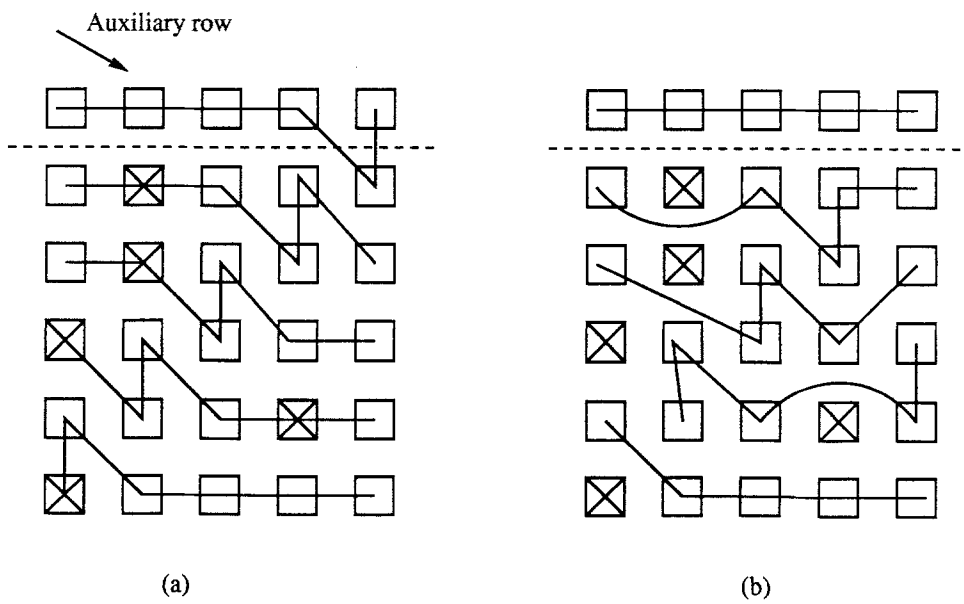



*Fig. 2.* Reconfiguration for Special Case 2.

The algorithm is illustrated in the 3 × 6 array of figure 1(b). For logical column one (which is a special column with degree two), we use two PEs from physical rows one and two and one PE from every other physical row. Since logical column two is not a special column, we only use one PE from every physical row. Logical column three is a special column with corresponding degree one and thus we use two PEs from physical row three and one PE from each of the rest of the physical rows. For the rest of the logical columns (which are not special columns), we use one PE from each physical row.

### *2.3. Reconfiguration for the General Case of an $N \times (K + 1)$ Physical Array with N Faulty PEs into a $N \times K$ Healthy Logical One*

In this section, we shall introduce an algorithm for reconfiguring arrays with general faulty patterns.

*Definition 4.* A faulty stack is defined to be a set of faulty PEs (in the physical array) that are in the same column and in consecutive rows. The size of a faulty stack is defined to be its cardinality.

*Definition 5.* Overlapping parts of two stacks are defined to be the parts of the stacks that are in the same rows. The size of the overlapping is defined to be the length of the overlapping parts.

Figure 3(a) illustrates an example of two overlapping stacks; each stack is of size 4 and they overlap completely. *The pattern of multiple overlapping faulty stacks of PEs is a very difficult case of faulty patterns to handle.* For example, as shown in Section 4, the algorithm in [8] fails whenever there are stacks that overlap by more than 1 PEs. It is easy to see why overlapping stacks is a hard pattern to reconfigure: every row of the array has only one spare PE available for the replacement of possible faulty PEs along the row. So, in the case of multiple overlapping stacks, only one of the stacks can be reconfigured using the spare PEs that are available in the same physical rows where the stacks overlap. The rest of the overlapping stacks must be reconfigured by spare PEs that are in rows above or below the rows where the overlapping occurs. The bigger the overlapping size and the larger number of overlapping stacks, the more difficult it is to *move* spare PEs from rows where they are available to the rows where the faulty PEs appear. *Our algorithm is designed*

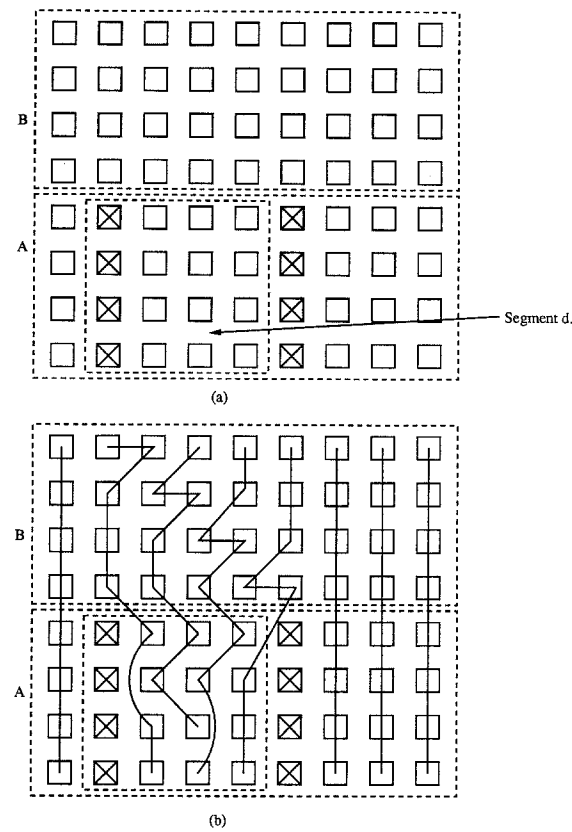



*Fig. 3.* Example: Reconfiguration for overlapping stacks.

to handle efficiently these hard cases of overlapping stacks locally so as to minimize the requirement of length of links between the PEs as much as possible.

*Example.* We will first describe the basic concepts of the reconfiguration procedures by introducing an example: Consider the array of figure 3(a) where an overlap of size 4 is taking place between two stacks of faulty PEs. The reconfiguration procedure can be summarized in two steps:

1. Consider for the moment only the rows where the overlapping is taking place; this is indicated by subarray $A$ in the figure. *There are two columns of faulty PEs in subarray A, but there is only one column of spare PEs available in the subarray.* So, only one of the two columns of the faulty PEs can be reconfigured using the spare PEs available in $A$. To reconfigure the other column of faulty PEs we use the healthy PEs in $A$ to *create* one more logical column at the expense, of course (since the total number of available PEs remains the same), of the length of some of the resulting columns. In other words,

we manage to create an extra column in $A$ by *stealing* PEs from the already existing columns of $A$. By doing so we manage to *move the need of PEs from the interior of subarray A to its boundaries, and thus closer to the rows where there are available PEs.* The way we create this extra column is as follows: we define a $4 \times 4$ segment $d$ in subarray $A$ by considering 3 fault free columns around one of the two overlapping parts of stacks as shown in figure 3(a); in general, we consider such segments for all but one of the overlapping parts of the stacks. Using the reconfiguration algorithm for Special Case 2 we can reconfigure the $4 \times 4$ segment that has one fault column (and only one faulty PE in each of its rows), into a $3 \times 4$ healthy segment. For the reconfiguration of the remaining faulty stack we are going to use the spare PEs that are available in the rows where the overlapping occurs. As shown in figure 3(b), an extra logical column has been created in subarray $A$ at the expense of the length of logical columns 2, 3, 4 and 5. These columns are now of length 3 instead of 4; that is the same as the length of the rest of the columns of subarray $A$.

2. Consider now the fault-free rows of the array indicated as subarray $B$ in figure 3(a). This fault-free block has an extra column of available PEs. We use these spare PEs to increase the length of some of its logical columns. More precisely, we will use the extra column of PEs in subarray $B$ to increase the length of the logical columns that have become *shorter* in subarray $A$ in the previous step. By doing this we manage to *move* the *availability* of PEs from the interior of the fault-free block to its boundary and thus bring the available PEs *closer* to the rows where they are needed. The way we do this is as follows: using the reconfiguration algorithm for Special Case 3 we reconfigure the fault-free block $B$, considering as special columns $c_i$'s the logical columns that have become *shorter* in subarray $A$. After the reconfiguration, the special columns of subarray $B$ increase in length to compensate for the loss of length that they suffered in subarray $A$ (see figure 3(b)).

Thus, the premise behind this reconfiguration algorithm is the local treatment of the blocks that have overlapping stacks and the local treatment of the fault-free blocks. We segment the array into subarrays that we can deal with according to the Special Cases presented earlier. We increase the number of columns when necessary at the cost of their length and we increase the length of the columns when necessary at the cost of their number.

The general way of reconfiguring the subarrays that have multiple overlapping stacks of faulty PEs in terms of the number of their columns can be described as follows:

***2.3.1. Multiple Stack Reconfiguration Procedure.*** Consider a subarray of size $s \times (N + 1)$ with $n$ faulty overlapping stacks of size $s$ each. The objective is to *create* $(n - 1)$ additional logical columns in the subarray, of course at the expense of the length of some of them. This is done as follows:

1. Define $(n - 1)$ non-overlapping segments $d_k$ $k = 1, \ldots, (n - 1)$, of size $s \times s$, each of which contains one faulty stack; for example, segment $d_{11}$ in subarray $ssa_1$, and segment $d_{21}$ in subarray $ssa_2$ as shown in figure 4(a). If such segments cannot be defined, divide the subarray into subarrays of smaller $s$ and apply the multiple stack reconfiguration procedure for each of them.
2. Reconfigure these $s \times s$ segments into $(s - 1) \times s$ fault-free logical segments, according to Special Case 2 (see figure 4(b)).

***2.3.2. The General Reconfiguration Procedure.***

*Step 1.* Partition the array into the following kind of subarrays that have $(N + 1)$ columns (see figure 4(a)):

1. Fault-free subarrays $ffsa_i$, $i = 1, \ldots, k$ of size $f_i \times (N + 1)$ each (see for example $ffsa_1$ and $ffsa_2$ of figure 4(a)).
2. Subarrays that have only one faulty PE in each row, $ofsa_i$, $i = 1, \ldots, l$ of size $o_i \times (N + 1)$ each (see for example $ofsa_1$ of figure 4(a)).
3. Subarrays $ssa_i$ of size $s_i \times (N + 1)$ that have $n_i$ $(> 1)$ stacks each of size $s_i$ (see for example subarrays $ssa_1$ and $ssa_2$ in figure 4(a); in $ssa_1$, there are two stacks of size 2, and in $ssa_2$ there are again two stacks but of size 1).

*Step 2.* For each subarray with faulty stacks $ssa_i$ apply the Multiple Stack Reconfiguration Procedure (see figure 4(a)(b)).

*Step 3.* Do the following:

1. Consider now the columns of each reconfigured segment, defined in Step 2, from left to right and the segments from left to right and from top to bottom.

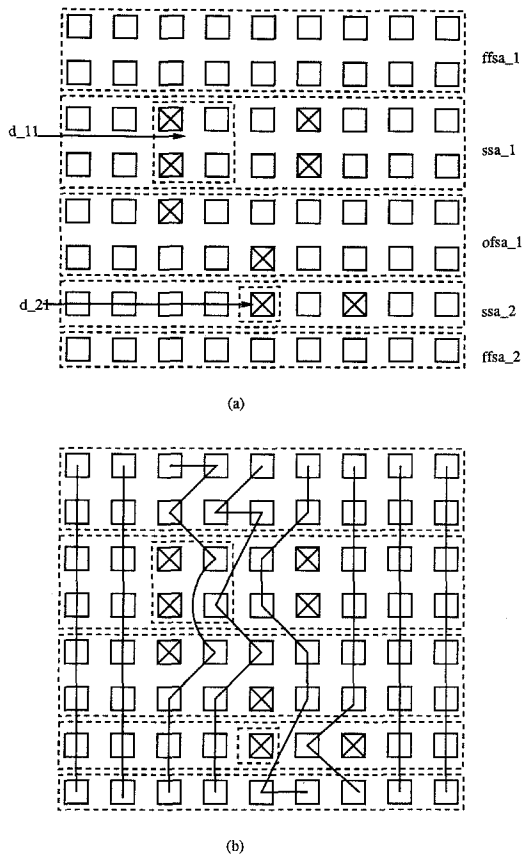



*Fig. 4.* Reconfiguration for the General Case.

Map each one of the columns of every segment to a faulty-free row considering the latter ones from top to bottom. For example, the first column of $d_{11}$ maps to physical fault-free row 1. The second column of $d_{11}$ is mapped to the physical fault-free row 2. The column of segment $d_{21}$ is mapped to the physical fault-free row 8.

2. Consider now every fault-free subarray $\text{ffsa}_i$. Let the columns $c_j$ of $\text{ffa}_i$ which are mapped to the rows of $\text{ffa}_i$ be the special columns of $\text{ffa}_i$. The degree $k_j$, corresponding to each *special* column $c_j$, equals the number of fault-free rows of the fault-free subarray $\text{ffsa}_i$ that are mapped to column $c_j$. For example, as shown in figure 4(b), $\text{ffsa}_1$ has special columns $c_1 = 2$ and $c_2 = 3$ with corresponding degrees $k_1 = 1$ and $k_2 = 1$; $\text{ffsa}_2$ has $c_1 = 5$ with corresponding degree $k_1 = 1$.
3. Reconfigure each subarray $\text{ffsa}_i$ according to Special Case 3 (see figure 4(b)).

The reconfiguration presented above can handle any faulty pattern if the number of the faulty PEs is no greater than the number of spare PEs, and achieves reconfiguration probability of 100%. In the case where the number of the faulty PEs is less than the number of spare PEs, we treat the healthy spare PEs of several fault-free rows as faulty and apply the general reconfiguration algorithm presented above. The above results can be summarized in the following theorem:

THEOREM 1. The reconfiguration algorithm presented above reconfigures any $N \times (K + 1)$ array with $N$ faulty PEs into a $N \times K$ healthy array of PEs.

*Proof.* On the proof of the theorem we shall make the following comments:

- The given array is segmented in Step 1 of the general reconfiguration algorithm into subarrays that: (1) are fault-free (2) have multiple faulty stacks (3) have only one column of faulty PEs.
- The subarrays that have only one column of faulty PEs get reconfigured by disregarding the column of faulty PEs. This faulty column is replaced by the column of spare PEs available in every subarray.
- In Step 2 of the general reconfiguration algorithm, we handle subarrays that have more than one ($n_i > 1$) faulty columns. We define segments around ($n_i - 1$) of the faulty columns of subarray $\text{ssa}_i$ and reconfigure them according to Special Case 2. Each segment gets reconfigured in terms of the number of the logical columns that it should have. The $n_i^{\text{th}}$ faulty column of the subarray gets replaced by the spare column of PEs available in the subarray. So the whole subarray $\text{ssa}_i$ gets reconfigured in terms of the number of its columns. Of course some of those columns become shorter in length than others. Step 3 compensates for this loss of length as discussed below:
- In the third step of the general reconfiguration algorithm, every column of the segments defined in Step 2 that need an additional PE to compensate for the length lost in Step 2 is mapped to a fault-free column; it is easy to see that the total number of those columns equals the total number of the fault-free rows. The logical columns that need additional PEs become the special columns of the fault-free segments with corresponding degree equal to the total number of additional PEs needed. Thus, we reconfigure the fault-free subarrays according to Special Case 3, increasing the length of the special columns by a number of PEs equal to their degree and reducing at the same time the total number of logical columns of the fault-free subarray by one.

## 3. Generalization for Arrays with More Than One Spare Columns

In this section, we shall outline the generalization of the reconfiguration algorithm presented in the previous section for arrays that have more than one spare column. The general idea for the reconfiguration remains the same when the number of spare columns is $M > 1$. It is still a local treatment of the faulty patterns that has segmentation of the array into different kinds of subarrays as its basic step. The basic differences compared to the algorithm presented in Section 2 can be summarized as follows:

- Instead of the segmentation of Step 1 of the general reconfiguration algorithm of Section 2, the array is now segmented into the following kind of subarrays: subarrays that (1) have less than $M$ faulty PEs in their rows, (2) have exactly $M$ faulty PEs in their rows and (3) have more than $M$ overlapping stacks of faulty PEs.
- The subarrays that have exactly $M$ PEs in their rows get reconfigured just by disregarding the $M$ faulty PEs of each row and substituting them with the $M$ spare PEs available in each row.
- The subarrays that have more than $M$ overlapping faulty stacks get reconfigured according to Step 2 of the general reconfiguration procedure presented in Section 2. Hence, we again reconfigure the subarray in terms of the number of its columns using the reconfiguration algorithm for special case 2 as we did when we had only one column of spare PEs. The only difference is that we need now only define segments around $(n_i - M)$ (instead of $(n_i - 1)$ in the case of the only one spare column) of the faulty stacks. This is because we need only *create* $(n_i - M)$ new logical columns since there are $M$ available columns of spare PEs to replace $M$ of the faulty stacks of the subarray.
- The subarrays that have less than $M$ faulty PEs in their columns stand for the fault-free subarrays of the general reconfiguration algorithm. Each one of their rows $r_i$ has $m_i$ faulty PEs, where $0 \leq m_i < M$, and thus has $M - m_i$ available PEs. So, in the third step of the general reconfiguration procedure, a total of $(M - m_i)$ columns (instead of just one as it was for the case of the only one column of spare PEs) of the segments defined in Step 2 are mapped to row $r_i$.
- The reconfiguration algorithm for the third Special Case is now simply generalized to an algorithm that is applied to arrays that have rows with less than $M$ faulty PEs instead of fault-free arrays. Each logical column can now get up to $M - m_i$ PEs from physical row $i$ that has $m_i < M$ faulty PEs.

## 4. Evaluation of the Algorithm

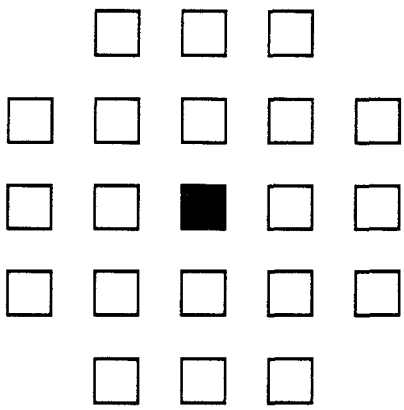
As already stated in Theorem 1, if the interconnect lengths between the logical neighbors of the reconfigured array are unrestricted, then our algorithm has a 100% reconfiguration probability for any general faulty pattern. The general objective for reconfiguration in models such as the one under consideration, however, is to derive a logical array such that the geometric distances between logical neighbors are kept small even for *difficult* faulty patterns. In this section we are going to evaluate the performance of our algorithm in terms of the interconnect lengths that it requires to reconfigure faulty patterns of certain difficulty. More precisely, in the first part of this section we are going to restrict the interconnect lengths allowed to be $\sqrt{5}$ or less, i.e., every PE is restricted to have all its logical neighbors among the 20 PEs that are shown in figure 5(a). The permitted set of logical neighbors for every PE will be defined as its *neighborhood*. We will prove that the basic reconfiguration algorithm and the reconfiguration algorithms for the special cases presented in Section 2 maintain the neighborhood constraints set above. We will also prove that several general faulty patterns can be reconfigured without increasing the neighborhood as depicted in figure 5(a). In Section 4.2, we use the results of Section 4.1 to show that our algorithm performs provably better than those in [8], i.e., can reconfigure many more faulty patterns than possible by the algorithm in [8]. We also show that for certain faulty patterns, our algorithm requires only constant interconnect lengths whereas, the algorithm in [2] requires lengths proportional to the size of the array.

### *4.1. Reconfiguration Under Neighborhood Constraints*

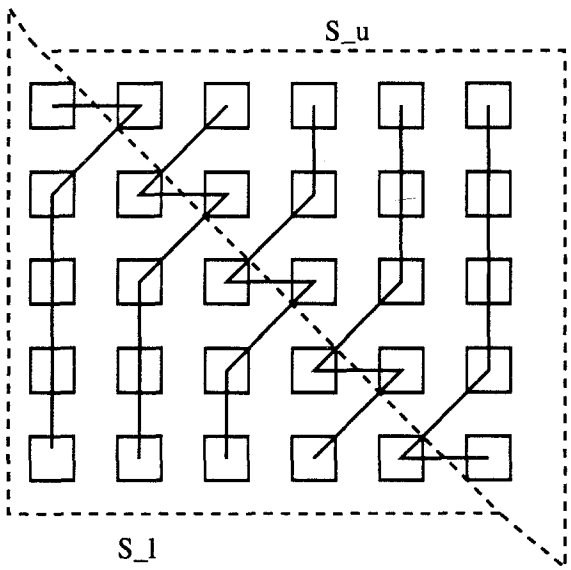
*Definition* 6. For each entry $(i, j)$ of the given array, we define as neighborhood (or legal neighborhood) of $(i, j)$ the set of PEs: $N_{i,j} = \{(k, l) / (k, l) = (i, j) + (m, n), (m, n) \in T\}$ where $T$ is a set of Tuplus.

In this section, we consider $N_{i,j}$ as shown in figure 5(a). In other words: $N_{i,j} = \{(k, l)/(i - k)^2 + (j - l)^2 \leq 5\}$. If the reconfiguration results in an interconnect scheme where all PEs are only connected with PEs that are in their neighborhoods (legal neighborhood) then the reconfiguration maintains the *neighborhood constraints.*

THEOREM 2. The reconfiguration for Special Case 3 maintains the neighborhood constraints when the degree $k_i$ of every special column $c_i$ of the array is less than or equal to one.



(a)




(b)

*Fig. 5.* Definitions.

*Proof.* The proof comes directly from figure 1(b). It is easy to check that for each entry $(i'j')$ of the logical array that is mapped in physical entry $(i, j)$ its logical neighbors: $(i' + 1, j')$, $(i' - 1, j')$, $(i', j' - 1)$, $(i', j' + 1)$ are mapped in $(i, j)$'s legal neighborhood.

THEOREM 3. The basic reconfiguration algorithm of a $(N + 1) \times N$ logical array from a $N \times (N + 1)$ physical array maintains the neighborhood constraints mentioned above.

*Proof.* The basic reconfiguration procedure is a restricted case of special case 3 when all the columns of the array are chosen to be special columns with corresponding degree one. So the proof follows immediately from Theorem 2.

We now prove that reconfiguration of Special Case 1 maintains the neighborhood constraints. We shall first make the following definitions:

*Definition 7.* $S_l$ is the subset of the PEs $(i, j)$ of the physical array for which $(i - j) \geq 0$ is true, (see figure 5(b)). The PEs of the array that are in $S_l$ from now on will be referred to as $S_l$ PEs.

*Definition 8.* $S_u$ is the subset of the PEs $(i, j)$ of the physical array for which $(i - j) > 0$ is true (see figure 5(b)). The PEs of the array that are in $S_u$ from now on will be referred to as $S_u$ PEs.

*Definition 9.* For each 2-fault logical column $i$ the corresponding 0-fault logical column is defined as the 0-fault logical column $j, j > i$ for which the following is true: all logical columns $k$, $i < k < j$ are 1-fault logical columns (see figure 1(c)).

We now present some lemmas that we will use later to prove that the reconfiguration algorithm for Special Case 1 maintains the neighborhood constraints.

COROLLARY 1. The number of 2-fault logical columns equals the number of the 0-fault logical columns.

LEMMA 1. The $S_l$ PEs of the $i^{\text{th}}$ logical column are part of the $i^{\text{th}}$ physical column whereas the $S_u$ ones are part of the $(i + 1)^{\text{th}}$ physical column.

*Proof.* Immediately from the basic reconfiguration algorithm. For example in figure 5(b) the $S_l$ PEs of the 3rd logical column belong to the 3rd physical column whereas the $S_u$ PEs belong to the 4th physical column.

LEMMA 2. Each logical column has at most two faulty PEs. If a logical column has two faulty PEs, then one of them is in $S_l$ and one in $S_u$.

*Proof.* The first part follows immediately from the basic reconfiguration algorithmn by observing that each logical column $i$ occupies parts of only two physical columns; the part of the physical column $i$ which is in $S_l$, and the part of the physical column $(i + 1)$ which is in $S_u$ (by Lemma 1). It is thus obvious that $i$ cannot have both faulty PEs in $S_l(S_u)$ since there is only one faulty PE in each of the physical columns $i$ and $(i + 1)$.

LEMMA 3. Suppose that the only faulty PE of logical column $i$ is in $S_l$. Then one of the following is true:

1. The $(i + 1)$ logical column has only one faulty PE which is also in $S_l$.
2. The $(i + 1)$ logical column has two faulty PEs (one in $S_u$ and one in $S_l$).

*Proof.* Since the only faulty PE of the logical column $i$ is in $S_l$, (in the $i^{\text{th}}$ physical column), the faulty PE of the $(i + 1)^{\text{th}}$ physical column should be in $S_l$. So the logical column $(i + 1)$ has one faulty PE in $S_l$. Therefore, if logical column $(i + 1)$ is 1-fault logical column, then 1 is true; otherwise (by Lemma 2) 2 is true.

LEMMA 4. Suppose that logical column $i$ has one faulty PE in $S_u$. Then one of the following is true:

1. The logical column $(i + 1)$ has only one faulty PE in the $S_u$ part of physical column $(i + 2)$.
2. The logical column $(i + 1)$ is 0-fault logical column.

*Proof.* Since there is a fault in the $S_u$ of physical column $(i + 1)$, there is no faulty PE in $S_l$ in the same physical column. So, if physical column $(i + 2)$ has a faulty PE n $S_u$, then 1 is true; otherwise 2 is true.

LEMMA 5. Suppose that logical column $i$ is fault free. Then one of the following is true:

1. The $(i + 1)$ logical column has only one faulty PE in $S_l$.
2. The $(i + 1)$ logical column is a 2-fault logical column.

*Proof.* Since $i$ is a 0-fault logical column, the $S_u$ part of $(i + 1)$ physical column is fault-free. Thus there is a fault in the $S_l$ part of the $(i + 1)^{\text{th}}$ logical column. So, if the $(i + 2)^{\text{th}}$ physical column has no fault in $S_u$, then 2 is true; otherwise 1 is true.

LEMMA 6. Each 2-fault logical column $i$ has a corresponding 0-fault logical column $j$. Moreover, all the 1-fault logical columns $k$ with $i < k < j$ have their faulty PEs in $S_u$.

*Proof.* Since $i$ is a 2-fault logical column then (by Lemma 1) it has a faulty PE in $S_u$. Then by Lemma 4 the $(i + 1)$ logical column is (a) either fault-free, in which case we are done (b) or has one faulty in $S_u$, in which case the lemma has been proved for logical column $(i + 1)$. Logical column $(i + 1)$ now has one faulty PE in $S_u$ and so Lemma 4 can be applied again to prove that logical column $(i + 2)$ is either fault-free or has only one fault in its $S_u$ part. One can easily show by induction that all the logical columns between the $i^{\text{th}}$ and the $j^{\text{th}}$ ones are 1-fault logical columns with faulty PEs in their $S_l$ parts.

The above lemma implies that there are one or fewer borrowing processes (like the ones described in Special Case 1) between the 2-fault logical columns and the 0-fault logical columns going on along the same columns of physical row $N$ (since there are only 1-fault logical columns between every 2-fault logical column and its corresponding 0-fault logical column). Moreover, there are no faulty PEs along the physical rows where the borrowing process is taking place. We now use the lemmas developed above to outline the proof that the reconfiguration of Special Case 1 maintains the neighborhood constraints.

THEOREM 4. The algorithm for the Special Case 1, that reconfigures a $N \times (N + 1)$ physical array with $N$ faulty PEs that are in different physical columns, into a $N \times N$ logical array, maintains the neighborhood constraints.

*Proof.* We have only to prove that the neighborhood constraints are maintained along the logical columns and the rows of the reconfigured array.

- The fact that the neighborhood constraints are maintained along the columns of the logical array follows immediately from Theorem 2. The first step of the reconfiguration algorithm for Special Case 2 is just the basic reconfiguration algorithm. One can easily see in figure 1(c)(d) that when a fault appears in the *straight* part of a logical column like the one appearing in the third logical column, then the interconnect length increases from 1 to $\sqrt{2}$ which is still within permissible limits. If a fault appears in the *bent* part of the logical column like the one appearing in the second logical column of the array of the same figure, then the interconnect length decreases from $\sqrt{2}$ to 1.

- The proof that the neighborhood constraints are maintained along the rows of the reconfigured array where there is no borrowing process taking place is illustrated in figure 1(c) and (d). The worst case as far as connections between PEs of the same logical row are concerned appears when two adjacent logical columns have faulty PEs before and after their bent parts. For example, see adjacent logical columns three and four of figure 1(c). One can easily check that the neighborhood constraints are maintained between the neighboring PEs of the two logical columns. One can also check that the neighborhood constraints are maintained in all the other cases when the two adjacent columns both have their faulty PEs before or after their bent part (see for example adjacent logical columns 4 and 5 in figure 1(c)), or when one of the two adjacent columns is fault-free and the other has its faulty PE before or after its bent part (see for example logical columns 5 and 6 of figure 1(c)).
- The neighborhood constraints are also maintained along the rows of the array where the borrowing process is taking place. Note that by Lemma 6, *there is no fault in $S_l$ between the 2-fault column and the fault-free column,* and *no more than one borrowing process occurs along the same columns of physical row N.* So the example of figure 1(d) proves that in general, the neighborhood constraints are maintained along the rows of the array where the borrowing process is taking place, since any general borrowing process is no different than the one presented in figure 1(d).

THEOREM 5. The reconfiguration procedure for Special Case 2 maintains the neighborhood constraints.

*Proof.* The proof that the reconfiguration algorithm of Special Case 3 maintains the neighborhood constraints follows immediately from Theorem 4 above, since Special Case 2 is reconfigured using the reconfiguration algorithm of Special Case 2.

So far we have proved that the reconfiguration for Special Cases 1 and 2 and the Basic Reconfiguration Algorithm maintain the neighborhood constraints. We have also proved that Special Case 3 can get reconfigured in the legal neighborhood if the degree of the special columns does not exceed one. The following theorem proves that the General Reconfiguration algorithm results in arrays that do not violate the neighborhood constraints if Step 3 of the general reconfiguration algorithm does not result into special columns with degree more than one.

THEOREM 6. Our algorithm can reconfigure any faulty pattern within the legal neighborhood if step 3 of the general reconfiguration algorithm does not result into special columns with degree more than one.

*Proof.* The proof of this theorem follows from theorem 2 and 4 presented above as follows:

- The fact that the neighborhood constraints are maintained in the fault-free subarrays of the array defined in Step 1 of the General Reconfiguration Algorithm follows immediately from Theorem 2.
- It is easy to check in figure 4(b) that the neighborhood constraints are maintained in the subarrays that have only one faulty PE in every row.
- The subarrays that have more than one column of faulty PEs are divided into segments that are reconfigured according to Special Case 1. The reconfiguration of those segments maintain the neighborhood constraints as stated in Theorem 4. The neighborhood constraints are also maintained between adjacent such segments or between such segments and fault-free adjacent columns (arguments similar to those presented in the proof of Theorem 4 hold for this case as well, since the *boundary* columns of the adjacent segments can be seen as adjacent columns in the same segment).
- The neighborhood constraints are also maintained between adjacent subarrays, as can be easily seen in figure 4(b).

### *4.2. Comparing the General Reconfiguration Algorithm to Other Reconfiguration Algorithms*

The algorithm presented in [2] uses a similar model to the one we have discussed in this paper. It is a general algorithm that can handle various faulty patterns with good interconnect requirement performance. Nevertheless it sometimes requires interconnect lengths that are proportional to the size of the array whereas our algorithm reconfigures the same patterns keeping the required interconnect length constant. Such an example is presented in figure 7(a). The series of PEs indicated as $S$ in the figure can grow proportionally to the size of the array and so can the length of the interconnect link indicated as $L$ in the same figure. Figure 7(b) shows the reconfiguration of the same faulty pattern according to our algorithm. It is easy to check that the interconnect length required remains constant as the size of the array grows.

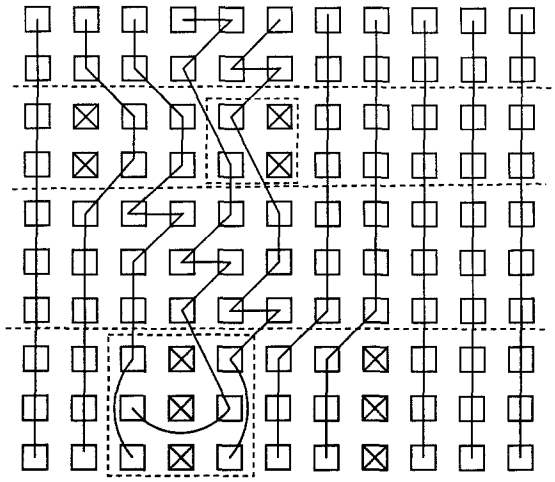


*Fig. 6.* Reconfiguration for the multiple-stack case.

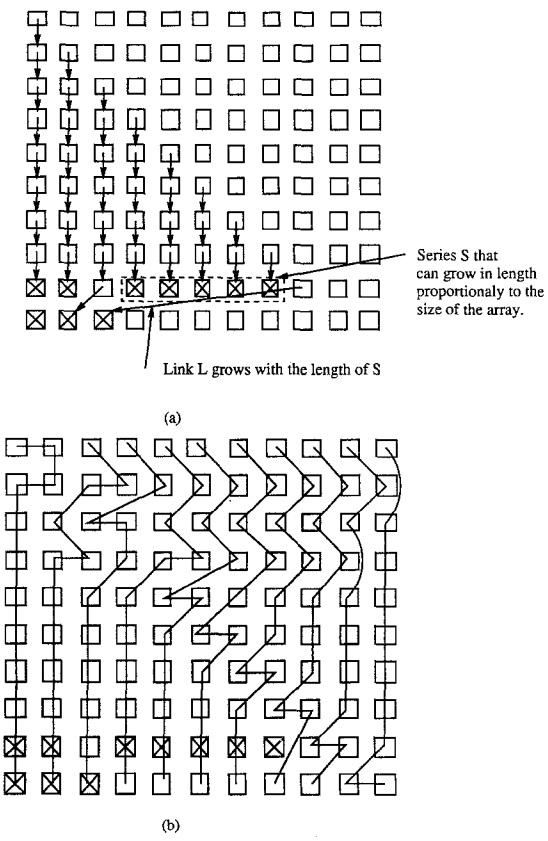



*Fig. 7.* (a) Reconfiguration according to the algorithm presented in [2]. (b) Reconfiguration according to our algorithm.

Another set of algorithms that uses the same kind of model as that which we have used is presented in [8]. The underlying model in [8] has only one column of spare PEs. The maximum length of interconnect links that is allowed is $\sqrt{5}$, and so the legal neighborhood used is the one that we studied in the Section 4.1. The algorithms presented in [8] are very simple, implementable in a distributed fashion, and achieve quite satisfactory reconfiguration yield. However, those reconfiguration algorithms are very fragile; they fail to reconfigure the array even when the faulty PE distributions are very simple. The following are only two simple instances (one can generate several other instances) where they fail: (1) if the bottom row and the top row does not have any faulty processors, and (2) if there is an overlap of faulty stacks even of size two.

The algorithms that we develop can reconfigure *all instances* of faulty arrays as the algorithms reported in [8]. Moreover, we can also reconfigure arrays with several other faulty distributions, *without increasing the neighborhood constraints*. In particular, our algorithm can reconfigure most of the overlapping stack patterns, keeping the size of the neighborhood as in [8] (see for example the reconfiguration of the case of multiple stacks in figure 6) and has no constraints about the first and the last rows of the array being fault-free. We now show that all the faulty patterns that are reconfigurable using the algorithm in [8], meaning the faulty patterns with overlapping stacks of size at most one can be reconfigured by our algorithm without increasing the interconnect length requirements.

When there are no overlapping stacks in the faulty array, then according to Theorem 4, the array is reconfigurable within the legal neighborhood. We now prove that when there are overlapping stacks of size at most one, we can reconfigure the array within the legal neighborhood. This completes the proof that our algorithm can reconfigure *all the faulty patterns as can the algorithms in [8] with no increase of the legal neighborhood.*

THEOREM 7. The general reconfiguration algorithm can handle overlapping stacks of size at most one while maintaining the reconfiguration constraints.

*Proof.* Figure 8 presents the worst case of overlapping of size one between stacks; the overlap occurs in adjacent columns of the array, and as such is supposed to be the worst case as far as interconnection between PEs in adjacent rows is concerned. Neither can the interconnection between PEs in the same logical column get harder. It is easy to check in figure 8 that the neighborhood constraints are maintained.

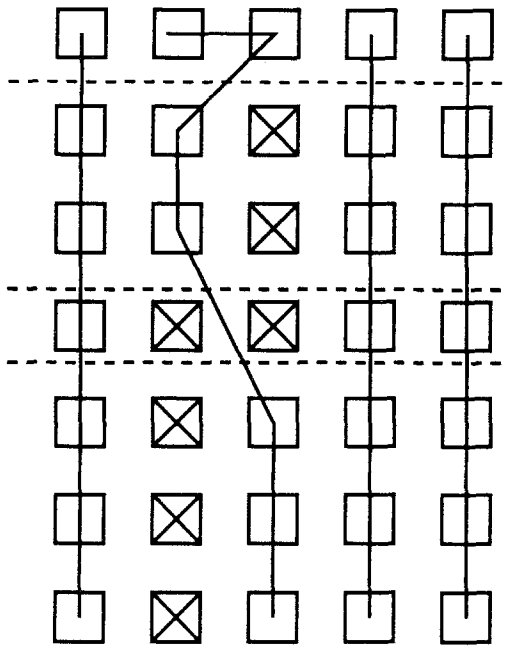


*Fig. 8.* Reconfiguration for worst case of size-one overlapping stacks.

## 5. Conclusion

In this paper we have presented new algorithms for reconfiguring processor arrays in the presence of faulty PEs. We used a model of a rectangular array with spare columns on one side and have focused on the minimization of the resulting lengths of the links between the PEs in the reconfigured array. Our algorithm is based on a local treatment of the faulty patterns and on an equal treatment of the spare and the non-spare PEs. We not only achieve a reconfiguration yield of 100%, but we can also reconfigure all the faulty patterns that other algorithms can reconfigure, and even more complicated ones, while keeping the required length of interconnection links the same or even less. Finally, we may note that the basic principles of our algorithm can be used to reconfigure processor arrays that have both columns and rows of spare PEs. Moreover, the algorithms presented for the special cases can be applied to change the *shape* of a rectangular array of given dimensions into a rectangular array of arbitrary dimensions, keeping the necessary interconnections as small as possible.

## Appendix

### A. Complexity Issues

#### *A.1. Complexity of the Reconfiguration Algorithms for the Special Cases*

All the reconfiguration algorithms for the following special cases are of linear time complexity ($O(N)$ where $N$ is the size of the array).

- Basic Reconfiguration Procedure (Section 2.1).
- Special Case 1 (Section 2.2); step 1 of the algorithm (in page 6) is of $O(N)$ complexity, each iteration of steps 2, 3, and 4 is of $O(1)$ complexity and the total number of iterations is $N$.
- Special Case 2 (Section 2.2); step 1 of the algorithm (in page 6) is of $O(1)$ complexity, step 2 is of $O(N)$ complexity, and step 3 is of $O(1)$ complexity.
- Special Case 3 (Section 2.2); both steps 1 and 2 are of linear complexity.

#### *A.2. Complexity of the Multiple Stack Reconfiguration Procedure*

We are now going to describe in detail a possible implementation of the multiple stack reconfiguration algorithm which is of complexity $O(N\ s)$ ($N$ is the number of columns of the subarray and $s$ is the number of rows). More precisely we are going to give an $O(N\ s)$ algorithm for implementing step 1 of the reconfiguration procedure described in Section 2.3 First we describe subroutine *SEGMENT* which defines (if possible) $(n - 1)$ non-overlapping segments of size $s \times s$ around $(n - 1)$ out of the $n$ faulty columns $c_1, \ldots, c_n$ of the $x \times (N + 1)$ subarray.

*SEGMENT:*
For subarray $ssa_j$ with faulty columns $c_1, \ldots, c_n$ do:

1. Initialize all the columns of the subarray to *UNMARKED*
2. *For* $i = 1$ to $n$ do
   Consider faulty column $c_i$. Assign (if possible) unmarked columns $(c_i - l, \ldots, c_i, \ldots, c_i + s + l)$ to segment $d_{ji}$ for the biggest possible $l$.

- If NOT possible and FLAG=true then:
  FLAG=false
  /*column $c_i$ is going to be the one for which we are not going to define a $s \times s$ segment*/
- If NOT possible and FLAG=false then:
  Signal *FAILURE*
  /*there is already a column for which we did not define a segment and we cannot define a segment for $c_i$ either */
- If possible then *MARK* columns ($c_i - l, \ldots, c_i, \ldots, c_i + s - l$) and continue.

Now the procedure that implements step 1 of the Multiple Stack Reconfiguration Algorithm can be described as follows:

Step 0: $i$=0

Step 1: Divide the subarray into two subarrays, one containing the first $s - i$ rows and the other the bottom $i$ ones.

Step 2: Apply procedure *SEGMENT* to the top subarray.

- If *SEGMENT* fails then:
  $i \leftarrow (i + 1)$ and goto step 1
- If *SEGMENT* succeeds then: divide the bottom subarray into subarrays of size $(s - i)$ each (the last one might be of smaller size if $i/(s - i)$ is not an integer) and apply *SEGMENT* to each of them (note that *SEGMENT* cannot fail in this case)

It is obvious that *SEGMENT* procedure takes $O(N)$ time and is called at most $s$ times. So step 1 of the Multiple Stack Reconfiguration algorithm is of $O(N s)$ complexity. Step 2 of the algorithm is of $O(N)$ complexity so the total complexity is $O(N s)$.

### *A.3. Complexity of the General Reconfiguration Algorithm*

The General Reconfiguration Algorithm presented in section 2.3.2 is of complexity $O(N^2)$. The complexity of each step is:

Step 1: The partitioning of the array into fault-free, one fault per row, and multiple stack subarrays in step 1 of the algorithm can be done in $O(N)$ time. A simple implementation of such a partition can be described as follows:

1. $i = 1$
2. If row $i$ is compatible with the subarray under consideration (meaning if the subarray under consideration being extended by row $i$ remains of the same type) then extend the subarray by row $i$.
3. If row $i$ is not compatible with the subarray under consideration, then start a new

   - *fault-free subarray* if the row is fault-free
   - *one fault per row subarray* if the row has only one faulty PE.
   - *multiple stack subarray* if the row has more than one faulty PE.

It is obvious that the procedure above is of $O(N)$ complexity.

Step 2: As it follows from the previous section, the complexity of this step is

$$O\left(\sum_{i=1}^{\#\text{ of ssa's}} N s_i\right) = O(N^2)$$

Step 3: All the three substeps of Step 3 are of linear complexity.

So the total complexity of the general Reconfiguration algorithm is $O(N^2)$.

## References


1. R.M. Tanner, "Fault-Tolerant 256-K Memory Design," *IEEE Transactions on Computers*, vol. C-33, pp. 314–322, 1984.
2. M. Chean and J.A.B. Fortes, "The Full-Use-of-Suitable-Spares (fuss) Approach to Hardware Reconfiguration for Fault-Tolerant Processor Arrays, *IEEE Transactions on Computers*, vol. 39, pp. 564–571, 1990.
3. J.S.N. Jean, H.C. Fu, and S.Y. Kung, "Yield Enhancement for WSI Array Processors Using Two-And-Half-Track Switches," in *International Conference on Wafer Scale Integration*, San Francisco, CA, pp. 243–250, 1990.
4. S.Y. Kung, S.N. Jean, and C.W. Chang, "Fault-Tolerant Array Processors Using Single-Track Switches," in *IEEE Transactions on Computers*, vol. 38, pp. 501–514, 1989.
5. W.R. Moore, "A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield," *Proceedings of the IEEE*, vol. 74, pp. 684–698, 1986.
6. A.L. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Array of Processors," *IEEE Transactions on Computers*, vol. C-32, pp. 902–910, 1983.
7. V.P. Roychowdhury, J. Bruck, and T. Kailath, "Efficient Algorithms for Reconfiguration in VLSI/WSI Arrays," *IEEE Transactions on Computers: Special Issue on Fault Tolerant Computing*, vol. 39, pp. 480–489, 1990.

8. M. Sami and R. Stefanelli, "Reconfigurable Architectures for VLSI Processing Arrays," *Proceedings of the IEEE*, vol. 74, pp. 712-722, 1986.
9. T. Varvarigou, V.P. Roychowdhury, and T. Kailath, "A Polynomial Time Algorithm for Reconfiguring Multiple Track Models," to appear in *IEEE Transactions on Computers*.
10. Theodora A. Varvarigou, Vwani P. Roychowdhury, and Thomas Kailath, "Reconfiguring Arrays Using Multiple-Track Models," submitted to *IEEE Transactions on Computers*, 1990.

**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, India, 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 an assistant professor in the Electrical Engineering Department at Purdue University. His research interests include parallel algorithms and architectures, special purpose computing arrays and VLSI design, fault-tolerant computation and design and analysis of neural networks.

**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. degree from Stanford University, Stanford, California in 1989.

She is currently a Ph.D. student in the Electrical Engineering Department at Stanford University. Her research interests include parallel algorithms and architectures, fault-tolerant computation and parallel scheduling on multiprocessor systems.

**Thomas Kailath** 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 joined Stanford University as an Associate Professor of Electrical Engineering in 1963. He has been a Full Professor since 1968, 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.

Dr. Kailath has worked in a number of areas including information theory, communications, computation, control, linear systems, statistical signal processing, stochastic processes, linear algebra and operator theory; his recent research interests include array processing, fast algorithms for nonstationary signal processing, and the design of special purpose computing systems. He is the author of *Linear Systems*, Prentice Hall, 1980, and *Lectures on Wiener and Kalman Filtering*, Springer-Verlag, 1981. He has held Guggenheim, Churchill and Royal Society fellowships, among other, 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 in 1989 and 1991. He served as President of the IEEE Information Theory Society in 1975, and was awarded an honorary doctorate from Linköping University, Sweden, in 1990. He is a Fellow of the IEEE and of the Institute of Mathematical Statistics and is a member of the National Academy of Engineering.