DISTRIBUTED STRUCTURING OF PROCESSOR ARRAYS
IN THE PRESENCE OF FAULTY PROCESSORS

ISRAEL KOREN and IRITH POMERANZ

INTRODUCTION

Reconfigurability is a desirable feature of processor arrays. First, it allows the embedding of different logical structures in a given physical topology. Secondly, it enables the restructuring of the array upon failure of one or more of its processors, (e.g., Koren 1981).

There are two alternative approaches to the reconfiguration (as well as the testing) of processor arrays, the centralized approach and the distributed one. Both are reviewed and compared in Koren and Pradhan (1986). In this paper we concentrate on distributed algorithms for restructuring of arrays in the presence of faulty elements. Several such algorithms are reviewed by Moore (1986) distinguishing between structuring of one dimensional arrays on rectangular and hexagonal physical arrays, and the mapping of more complex logical structures like binary trees and two dimensional arrays.

Due to the higher complexity of the latter, schemes for introducing switching elements into the design of physical arrays have been proposed (e.g., Moore 1986). These schemes attempt to achieve 100-percent utilization of the fault-free processing elements (*PE* s) on one hand, and try to achieve a high degree of topological regularity in order to keep the reconfiguration problem within bounds.

A taxonomy for representing topologies of processor arrays using different types of *PE* s and switching elements has been suggested by Koren and Pradhan (1986). We consider here only one class of these topologies in which only *PE* s are employed. For this type of topologies the mapping of one dimensional arrays (in the presence of faults) is straightforward but the mapping of binary trees and two dimensional arrays is very complex. Consequently, a different approach has been suggested by Koren (1981) and by Gordon *et al* (1984). According to it, all *PE* s in the same row and column (of the physical array) as the faulty one are turned into connecting elements (*CE* s), i.e., elements that only pass on the messages they receive. The remaining array is smaller, but has the same configuration as the original one, so that the same structuring algorithms can be used. This simple approach is suitable when a small number of faults can be expected (e.g., Koren 1986) since a large number of faulty *PE* s may cause too many other elements to become useless.

We propose here a totally different approach which is applicable to structures in which every element communicates directly with a small number of neighbors relative to the number of neighbors in the physical array. It is presented in the next section through the mapping of a binary tree onto a rectangular or an hexagonal array. In the last section we describe additional physical arrays and logical structures this approach can be applied to.

THE BASIC PRINCIPLES OF THE NEW APPROACH

Initially, all the elements are either free or faulty. We assume that a testing phase takes place before reconfiguration begins (e.g., Koren 1981) and that faults don't occur during the reconfiguration phase. Therefore, all the neighbors of a faulty element know that it is faulty before the reconfiguration phase is initiated.

We assume the presence of a host that initiates the testing and then the reconfiguration phase by sending the appropriate messages to an element on the boundary of the array. After the reconfiguration phase ends, the host receives an acknowledgement (positive if the reconfiguration ends successfully and negative otherwise). In practice, since it can't be guaranteed that no failures would occur during the reconfiguration phase, and that an acknowledgement would reach the host, a time-out mechanism is to be used by the host. Another case where such a mechanism is necessary, is discussed later.

We further assume that the structure of the required binary tree is predetermined and that an element which receives a structuring message knows in what directions its successors should be and what its structuring messages to them should contain. Two of the ways to achieve this are:

(1) The tree has a regular structure, like an H-tree (e.g., Koren 1981), and each element has a copy of a routine for calculating the directions of its successors and the messages it should send them.

(2) The structure is given by a list which is passed on from one element to its successors while mapping the tree, as in Gordon *et al* (1984).

As in the simple approach presented in Koren (1981), the only modification made by our approach to the structure of the tree is the addition of CEs in order to bypass faulty elements. However, in contrast to the approach in Koren (1981), CEs are added here only where they are needed. Consequently, a substantially smaller number of fault-free elements are turned into CEs, leaving a larger number of usable elements.

The main idea behind our approach can be simply stated as follows : when a subtree can't be mapped in its original position because of faulty elements, it is 'moved' in a predetermined direction, until it can either be mapped successfully, or can't be moved any more. The direction we have chosen (for reasons to be later explained) is defined as follows: Let e be an element of the array, which is to be a PE in level i, $(i > 1)$, of the tree. Let $p(e)$ be e's predecessor. Then, if one of e's subtrees (of level i-1) can't be mapped in its original place, it is moved in direction $d(p(e),e)$, which is the direction from $p(e)$ to e.

As an example to the way the algorithm works, assume that a 3-level H-tree is to be mapped onto a 4 x 5 rectangular array, as in Figure 1. In this figure, the number under the 'PE' indicates the level of the PE within the H-tree. Assume also that the array contains a faulty element, marked by an X in Figure 2(a). Then, the subtree marked S_1 in Figure 1 is mapped in its original position since no faulty element prevents its mapping. Subtree S_2 can't be mapped in its original place, and it has to be moved in the direction from CE to PE_1. For this purpose another CE, marked CE_1 in Figure 2(a), is added, and S_2 is to be mapped starting from it as if it were PE_1. Now, S_3 can't be mapped in its original position relative to PE_2, and so CE_2 is added, as shown in Figure 2(b), and the mapping of the tree is now complete.

As another example, consider the fault pattern in Figure 3(a) and the mapping of a 3-level H-tree as in Figure 1. S_5 is moved by adding CE_2. Another CE, CE_1, is added because S_2 can't be mapped in its original place. This appears in Figure 3(b). Neither S_3 nor S_4 can be mapped starting from PE_2 in Figure 3(a), so it is turned into a CE, CE_3 in Figure 3(b), allowing the mapping of S_2.

Note that the minimal dimension of a rectangular array that would contain a 3-level H-tree is 3 x 3. Using the approach presented in Koren (1981) in the last example, the remaining array after turning elements into CE's is too small to contain a 3-level H-tree. Using our approach, the required array dimension is 3 x 4.

The connecting elements in the examples above and in our algorithm in general, are of two types :

(1) Connecting elements that pass on the messages they receive in the same direction. Elements of this type are part of the structure of an H-tree, like the only CE in Figure 1. They are also used in Koren (1981) for bypassing faulty elements. An example for their use in our approach is CE_3 in Figure 3(b). This type will be denoted CE.

(2) Connecting elements that pass on the incoming messages in a direction different from the one they were received from. They will be called Switching elements (SEs). In our algorithm, an SE is used to connect a PE that has only one successor in its original place, to its other successor. CE_1 in Figure 2(b) as well as CE_1 and CE_2 in Figure 3(b) are SEs.

Three types of messages are passed during the mapping of the tree :

(1) Structuring messages which are sent from an element to its successors, by which they receive the following information : (a) That a binary tree is to be mapped, (b) Their level in it, (c) Their type: PE, CE or SE, and (d) Other details which depend on the way the structure of the required tree is given.

(2) Acknowledgement (ACK) messages, which are sent from an element to its predecessor once the mapping of the subtree starting from it is completed. ACK is positive if the mapping ends successfully and is negative otherwise.

(3) Cancellation messages which are sent from an element to one of its successors when the subtree starting from the latter is to be canceled. The need to cancel a subtree arises when it has been determined that the mapping of the subtree in its current position can not be successfully completed. The purpose of cancellation is to free elements (PEs, CEs and SEs) that may be useful in the following stages of mapping.

The first two types of messages are used by all 'conventional' distributed structuring algorithms. The third type is unique to our approach, that treats faulty elements locally, trying to bypass them by modifying the smallest

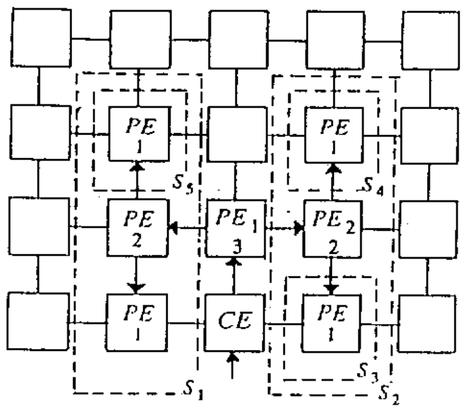
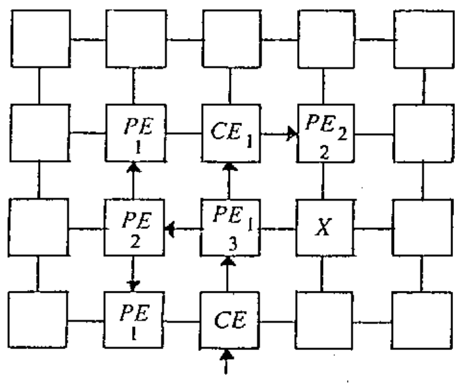
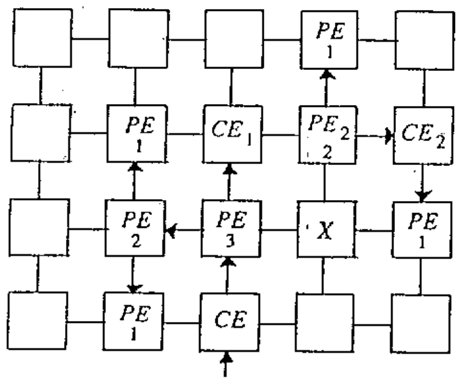


Figure 1: A 3-level H-tree on a rectangular array.

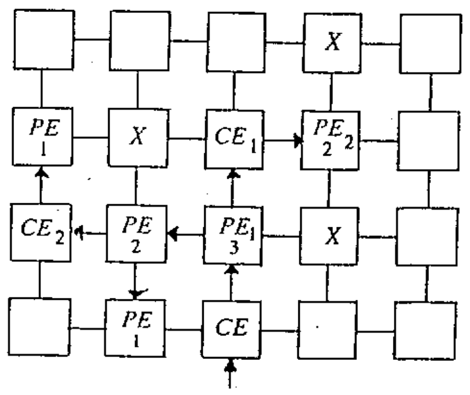


(a)

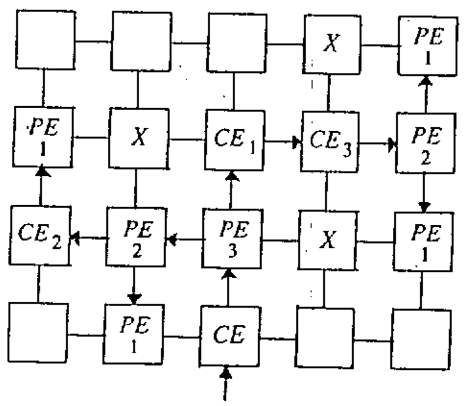


(b)

Figure 2: Mapping a 3-level H-tree in the presence of one fault.



(a)



(b)

Figure 3: Mapping a 3-level H-tree in the presence of three faults.

possible subtree.

In the following subsection we present a general description of the steps element e executes for every type of message it may receive. The detailed algorithm can be found in Appendix A. We concentrate on the messages that are passed and ignore book-keeping details like updating the directions of the successors and predecessor, the level, the type, etc.

General description of the algorithm

When element e receives a structuring message by which it should become a PE in level i ($i>1$), it attempts to map two (i-1)-level subtrees according to the structure of the required tree. In case one or both of e's subtrees can not be mapped in its original direction, the alternative direction we have chosen for moving a subtree is $d(p(e),e)$ defined above.

If both e's successors are in directions different from $d(p(e),e)$ and both are non-faulty, e sends the appropriate structuring messages to both and waits for acknowledgement. If it receives positive ACKs from both of them, it sends a positive ACK to its predecessor. If one of them returns a positive ACK and the other returns a negative ACK (or the other successor is faulty), e tries to map the missing subtree using the element in direction $d(p(e),e)$ as an SE. If this element is fault-free, e sends it a structuring message instructing it to become an SE and map the missing subtree. If it returns a positive ACK, e returns a positive ACK to its predecessor. The case where it returns a negative ACK is described later. If the element in direction $d(p(e),e)$ is faulty, e sends a cancellation message to its existing subtree and sends a negative ACK to its predecessor. If both e's successors return a negative ACK (or both are faulty), e becomes a CE, and tries to move the whole i-level subtree in direction $d(p(e),e)$. If the element in that direction is fault-free, e sends it the structuring message it has received. Otherwise it sends its predecessor a negative ACK.

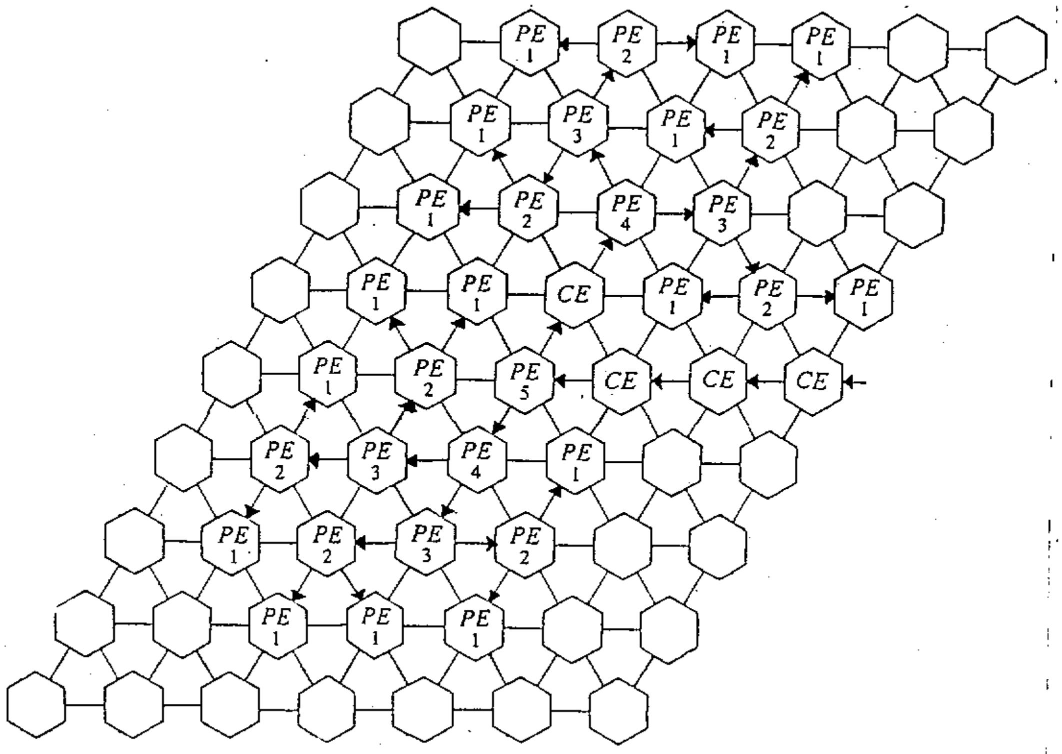
If one of e's successors is in direction $d(p(e),e)$, e sends a structuring message only to the other one, and waits for an ACK. If a negative ACK is received, e becomes a CE and the element in direction $d(p(e),e)$ is used instead of e to map the i-level subtree. If a positive ACK is received, e sends the appropriate structuring message to its other successor. This is not done earlier in order to avoid the need to cancel the subtree if the mapping of the other one fails. There are two possible cases in which e may receive a negative ACK from its successor in direction $d(p(e),e)$. The first is that there are too many faulty elements in that direction. The second is that the other subtree occupies elements that are necessary for the mapping of the first. A solution for both cases is to cancel the existing subtree, then turn e into a CE and move the whole i-level subtree in direction $d(p(e),e)$.

Example: A five level subtree that was suggested by Gordon *et al* (1984) for mapping a binary tree onto an hexagonal array is repeated in Figure 4(a). The steps of our algorithm for this structure, with the fault pattern of Figure 4(b) are as follows. When PE_1 in Figure 4(b) receives a structuring message, it sends two structuring messages to its two successors. It receives a negative ACK from its left successor because of the faulty element, and a positive ACK from its right successor. PE_1 sends then a structuring message to F_1 (F denotes a free element) in Figure 4(b), instructing it to become a SE and to map the missing left-subtree. The mapping goes on in the usual manner until the stage that appears in Figure 4(c) : PE_3 sent a structuring message only to its right successor, since its left successor is faulty. It received a positive ACK, so it sent a structuring message to the element in direction $d(p(PE_3),PE_3)$, instructing it to become an SE and to map the missing subtree, but received a negative ACK since that element is already a part of the tree. PE_3 sends a cancellation message to its right successor, becomes a CE and sends $d(p(PE_3),PE_3)$ a structuring message identical to the one it has received. The final result appears in Figure 4(d).

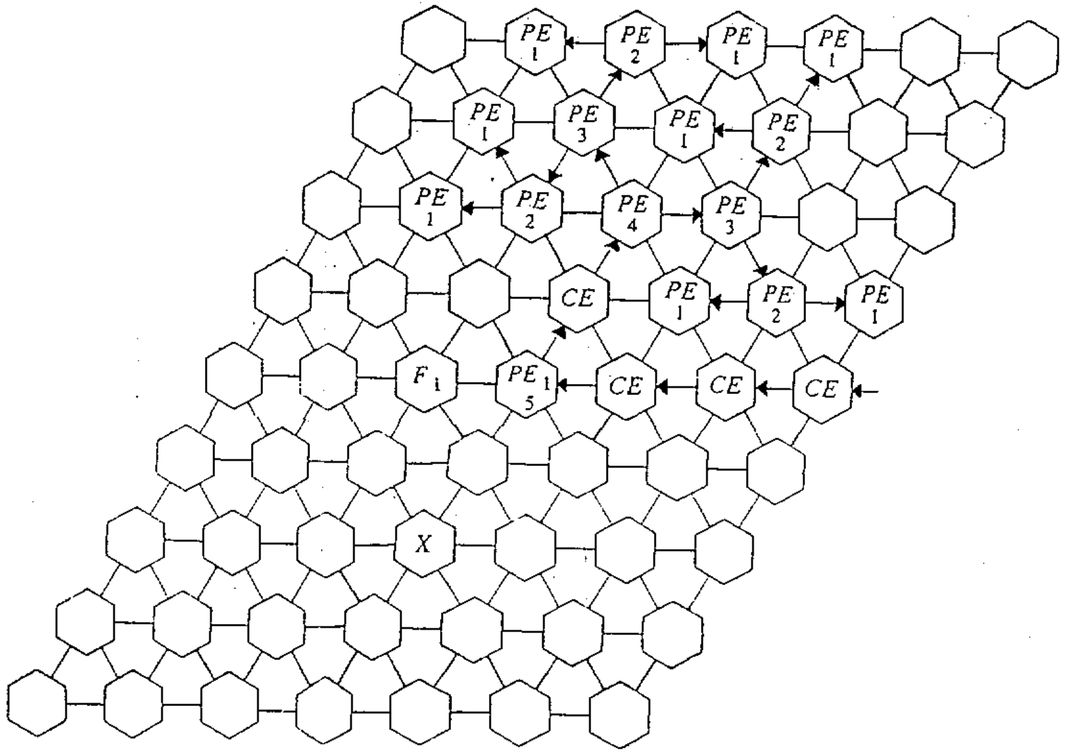
SIMULATION AND STATISTICS

We have programmed the above algorithm in 'C', in two versions. The first simulates the mapping of an H-tree onto a rectangular array. The second simulates the mapping of a binary tree defined by any given list, onto an hexagonal array. Both versions are sequential and simulate in a conventional manner the situation where many processors work in parallel. Although the number of possible mappings the algorithms try is finite, it may become very large for certain fault patterns. We have therefore, limited the number of time units it takes to reconfigure the array. (A time unit is the time it takes an element to execute the algorithm once). In practice, it would be up to the host to stop the reconfiguration when a certain time limit is exceeded.

To evaluate the efficiency of our approach, relative to the approach presented in Koren (1981), we have compared the percentage of successful mappings as a function of p, the probability of an element to be faulty, for a

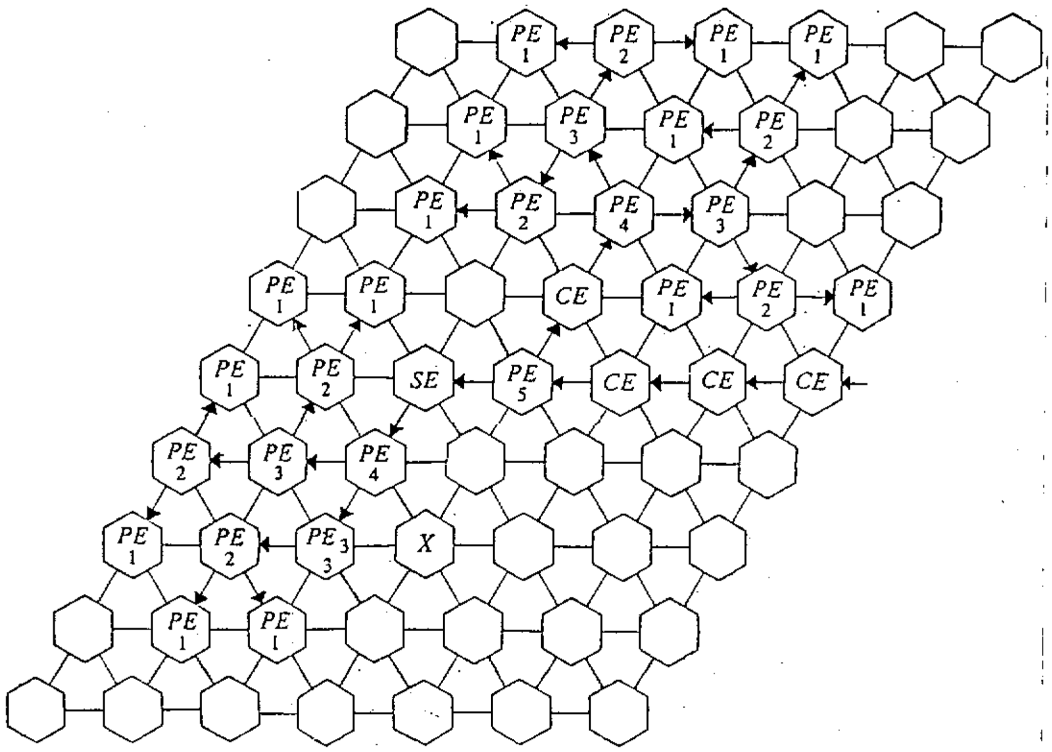


(a)

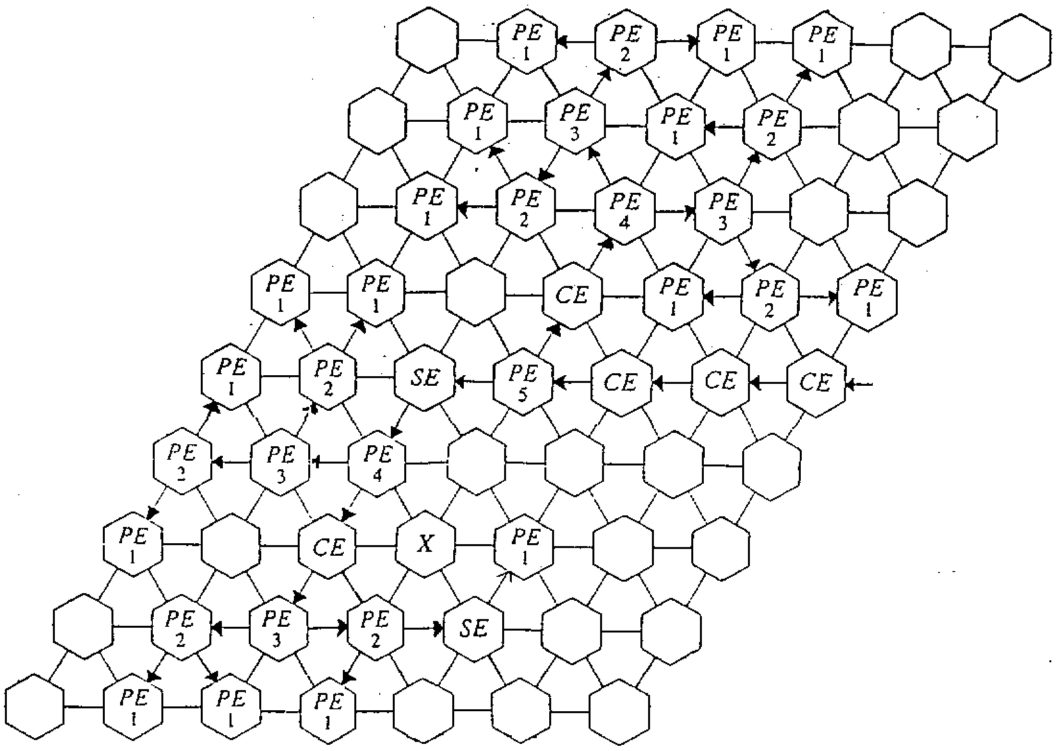


(b)

Figure 4: Mapping a 5-level binary tree on an hexagonal array.



(c)



(d)

Figure 4 (Cont'd): Mapping a 5-level binary tree on an hexagonal array.

6-level H-tree on a 10 x 18 rectangular array. The results are given in Figure 5 illustrating the advantage of our algorithm, whose corresponding curve is marked by (*i*).

Other criteria for comparing different structuring algorithms or different logical structures are: minimum array dimension required for mapping a given structure (as a function of p), maximum propagation delay between the host and the furthest element of the structure, maximum delay between logically adjacent elements, etc. As an example, we have compared two structures of a binary tree on an hexagonal array: (*i*) The subtree from Figure 4(a), and (*ii*) A 5-level H-tree. We have calculated for these two alternative structures (which are denoted by (*i*) and (*ii*), respectively), the percentage of successful mappings on a 11 x 10 array (see Figure 6) and the maximum delay between the host and the furthest leave (in Figure 7). All the results are averages over 100 random fault patterns. Structure (*i*) was shown to be more efficient than (*ii*) when being mapped on a fault-free physical array (Gordon *et al* 1984). This however, did not consider the effects faulty PEs may have. Figure 6 shows that (*ii*) has a higher probability to be successfully mapped in the presence of faulty PEs, possibly due to the lower chance that a local change in the mapping of (*ii*) may interfere with the whole mapping. Figure 7 shows that (*i*) is superior when propagation delays are considered even in the presence of faults.

EXTENSIONS AND CONCLUSIONS

The approach presented here can be extended in several ways. First, one should note that the algorithm as presented in this paper can handle also faulty connections. As in Koren (1981), we assume that during the testing phase, an element on one side of a faulty connection concludes that the element on the other side is faulty. The result is that faulty connections aren't used in the reconfiguration phase, as required.

Other, more complex extensions are possible. In order to keep our algorithm as simple as possible, we have chosen to use one alternative direction for moving subtrees that can't be mapped in their original places. It may be advantageous to use additional directions, as in the example of Figure 8, where an H-tree is to be mapped onto an hexagonal array. In the version of the algorithm presented above, PE_1 would return a negative ACK, and the mapping would fail. It is possible, though, to map the H-tree by using a direction different from $d(p(PE_1),PE_1)$, as in Figure 8.

Our algorithm can also be extended to other structures and other array topologies. The characteristic of a binary tree that makes it possible to map it using our approach, is that no connection is needed between two subtrees of the same level (at any level). This means that they can be mapped independently, and that local decisions on how to bypass faulty elements are possible. Any structure that has the same characteristic can be mapped using this approach. The only condition on the physical array topology is that it be possible to embed the given structure in it. For example, it should be possible to extend this approach to the mapping of a t-ary tree (not necessarily a complete one) onto an array where each element has at least $t+1$ neighbors. We could not extend our approach to the mapping of a rectangular array since it can not be divided into independent sub-arrays.

Appendix A

We use the following notation :
$d(p(e),e)$ - The direction to element e from its predecessor.
$dr(e)$ - The direction from element e to its right successor.
$dl(e)$ - The direction from element e to its left successor.
(Note that e's right or left successor is not necessarily its right or left neighbor).
$ds(e)$ - The direction from element e to its s-successor, s=right or s=left.
$R\_ACK(e)$, $L\_ACK(e)$ - indicate the type of ACK received by element e from its right, left successor.
$c(e,d)$ - a Boolean function which indicates whether element e has a non-faulty neighbor in direction d.

Structuring message(type,level) to a free element

```
if type=$PE$ and level=1 then $e$ sends a positive $ACK$ to its predecessor;
if type=$PE$ and level $\neq$ 1 then
  if $dl(e)=d(p(e),e)$ or $dr(e)=d(p(e),e)$ then
    if not $c(e,d(p(e),e))$ then $e$ sends a negative $ACK$ to its predecessor;
    /* one of $e$'s successors can't be mapped, and it can't be moved in direction $d(p(e),e)$ */
    if $dr(e)=d(p(e),e)$ then $s=l$;
    if $dl(e)=d(p(e),e)$ then $s=r$;
    if $c(e,d(p(e),e))$ and not $c(e,ds)$ then /* Both $e$'s successors must be moved in direction $d(p(e),e)$ */
      $e$ becomes a $CE$ and passes the structuring message it has received to the element in direction $d(p(e),e)$;
      if $c(e,d(p(e),e))$ and $c(e,ds)$ then
        e sends the appropriate structuring message to $ds$;
        it marks that no structuring message was sent to $d(p(e),e)$ by assigning '?' to $s\_ACK(e)$;
        /* No structuring message is sent to $d(p(e),e)$ until $ds$ returns an ACK */
    if $dl(e) \neq d(p(e),e)$ and $dr(e) \neq d(p(e),e)$ then
      for $s=l$ and $s=r$ do
        if $c(e,d(p(e),e))$=false and ( $c(e,dl(e))$=false or $c(e,dr(e))$=false ) then
          e sends a negative ACK to its predecessor;
        if not $c(e,ds(e))$ then $s\_ACK(e)$=negative;
        else e sends the appropriate structuring message to $ds(e)$;
      if $L\_ACK(e)$=negative and $R\_ACK(e)$=negative then
        e becomes a CE and passes the structuring message it has received to the element in direction $d(p(e),e)$;
  if type=CE then
    if $c(e,d(p(e),e))$ then e sends the appropriate structuring message to the element in direction $d(p(e),e)$;
    else it sends a negative ACK to its predecessor;
      if type=SE then
      /* either a right or a left subtree should be mapped from e, denote it an s-subtree (s=left or s=right) */
      if $ds=d(p(e),e)$ then
      if not $c(e,d(p(e),e))$ then e sends a negative ACK to its predecessor;
      else it sends the appropriate structuring message to $d(p(e),e)$;
    if $ds \neq d(p(e),e)$ then
      if not $c(e,d(p(e),e))$ and not $c(e,ds)$ then e sends a negative ACK to its predecessor;
      if not $c(e,ds)$ and $c(e,d(p(e),e))$ then
        e becomes a CE and passes the structuring message it has received to $d(p(e),e)$;
      if $c(e,ds)$ then e sends the appropriate structuring message to $ds$;
```

Structuring message to a *PE*, *CE* or *SE*

```
  e returns a negative ACK to the element that sent the message;    /* e is already part of the tree. */
```

ACK* message to a *PE

```
let s denote the direction from which the message was received;
$s\_ACK(e)$= the value of the ACK received;
if $R\_ACK(e)$=positive and $L\_ACK(e)$=positive then
  e sends a positive ACK to its predecessor;
if $s1\_ACK(e)$=positive and $s2\_ACK$=? and $s1 \neq s2$ then
/* This happens if $ds2=d(p(e),e)$ and means that no structuring message was sent to $ds2$ */
  e sends the appropriate structuring message to $ds2$;
if $s1\_ACK(e)$=negative and $s2\_ACK(e)$='?' then
/* $ds2=d(p(e),e)$ and the other subtree couldn't be mapped */
  e becomes a CE and passes the structuring message it has received to $d(p(e),e)$;
if $s1\_ACK$=negative and $s2\_ACK$=negative then
  if $ds1(e) \neq d(p(e),e)$ and $ds2(e) \neq d(p(e),e)$ then
    if $c(e,d(p(e),e)$ then
    e becomes a CE and passes the structuring message it has received to $d(p(e),e)$;
    else it sends a negative ACK to its predecessor;
if $s1\_ACK$=positive and $s2\_ACK$=negative then
  if $ds2(e)=d(p(e),e)$ then
  /* e has an s1-subtree and its s2-subtree couldn't be mapped in direction $d(p(e),e)$, e's i-level subtree is
moved */
    e sends a cancellation message to $ds1$;
    after DT time units /* long enough to let the s1-subtree cancel */ it becomes a CE and passes the structuring
message it has received to $d(p(e),e)$;
  if $ds1(e) \neq d(p(e),e)$ and $ds2(e) \neq d(p(e),e)$ then
    if $c(e,d(p(e),e))$ then e sends a structuring message to $d(p(e),e)$, instructing it to become a SE and to map
e's s2-subtree;
    else e sends a negative ACK to its predecessor;
```

ACK* message to a *CE

```
e passes the message on to its predecessor;
```

ACK* message to an *SE

```
/* $e$ is the root of an $s$-subtree, $s=l$ or $s=r$. */
$s\_ACK(e)$=the value of the $ACK$ received;
if $s\_ACK(e)$=negative then
  if $ds(e)=d(p(e),e)$ then $e$ sends a negative $ACK$ to its predecessor;
  if $ds(e)\neq d(p(e),e)$ then
    if $c(e,d(p(e),e))$ then
      $e$ becomes a $CE$ and passes the structuring message it has received to $d(p(e),e)$;
      else it sends a negative $ACK$ to its predecessor;
if $s\_ACK(e)$=positive then $e$ sends a positive $ACK$ to its predecessor;
```

Cancellation message to a *PE*
If e's level $\neq 1$ then e sends cancellation messages to both its successors;

Cancellation message to a *CE* or *SE*
e sends a cancellation message to its successor;

REFERENCES

Gordon D, Koren I and Silberman G M 1983 Fault Tolerance in VLSI Hexagonal Arrays, typescript, Dept. of Electrical Engineering, Technion

Gordon D, Koren I and Silberman G M 1984 Embedding Tree Structures in VLSI Hexagonal Arrays, IEEE Trans. on Comp. C–33 104-107

Koren I 1981 A Reconfigurable and Fault-tolerant VLSI Multiprocessor Array, Proc. 8th Annual Symp. on Comp. Arch. 425-442

Koren I 1986 Comments on "The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors," IEEE Trans. on Comp. C–35 93

Koren I and Pradhan D K 1986 Yield and Performance Enhancement Through Redundancy in VLSI and WSI Multiprocessor Systems, IEEE Proc. 74, 699-711

Moore W R 1986 A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield, IEEE Proc. 74, 684-698

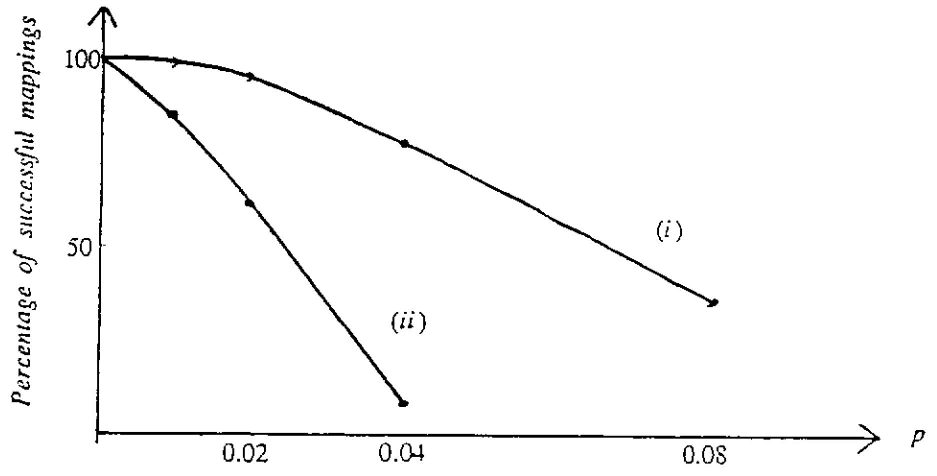


Figure 5: Percentage of successful mappings of an H-tree on a rectangular array as a function of p.

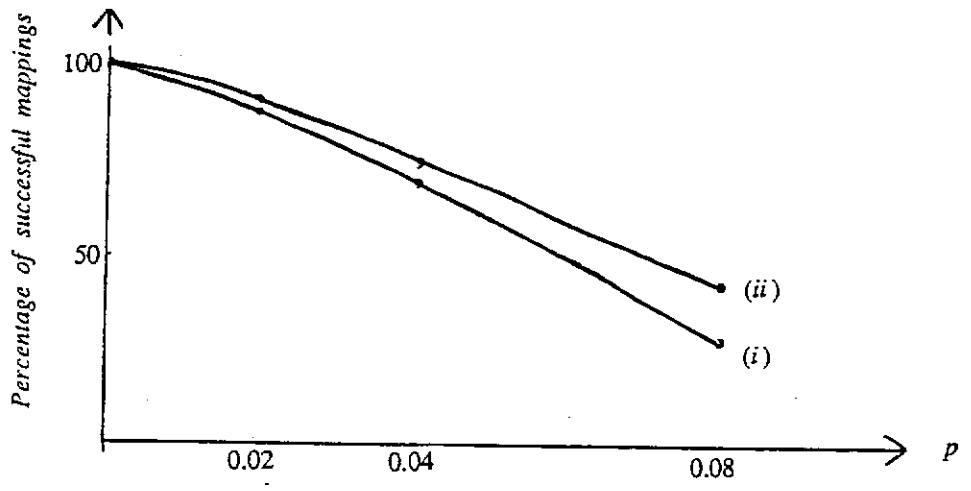


Figure 6: Percentage of successful mappings of two binary trees on a hexagonal array as a function of p.

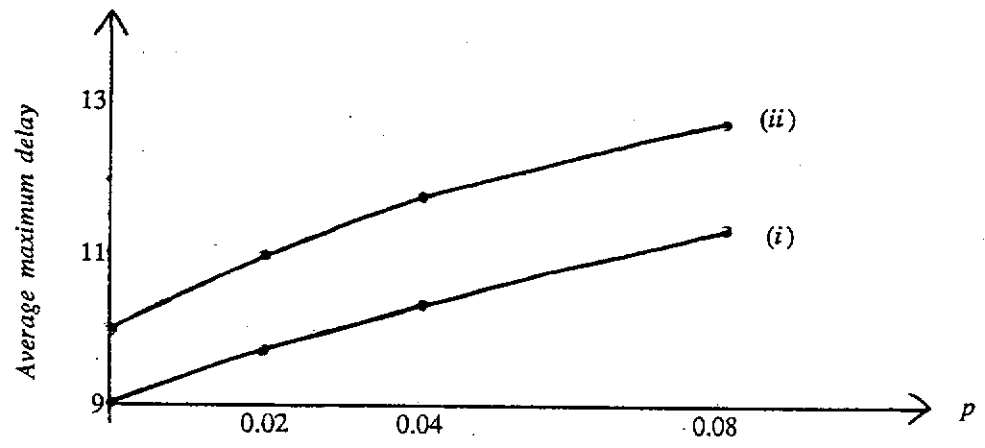


Figure 7: Propagation delays for the two binary trees (as in Figure 6) as a function of p.

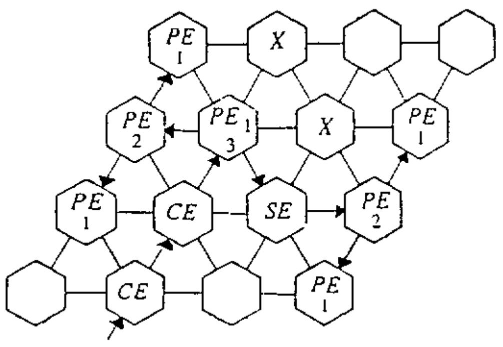


Figure 8: Using other alternative directions for moving sub-trees.