JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 12, 1–11 (1991)

A Random, Distributed Algorithm to Embed Trees in Partially Faulty Processor Arrays*

DIPAK SITARAM, ISRAEL KOREN, AND C. M. KRISHNA

Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Massachusetts 01003

A random and distributed, yet simple, algorithm is presented to embed trees in rectangular processor arrays in the presence of faults and manufacturing defects. One major characteristic of the algorithm is its ability to obtain embeddings in the presence of defect clusters in the processor array. Another useful feature is that the algorithm can be executed on the processor array itself, requiring only limited interprocessor communication. The performance of random algorithms is very difficult to analyze. Therefore, extensive simulation experiments were conducted and are presented in this paper. We also suggest an approach to overlapping algorithm runs to reduce the time needed to obtain a good embedding. The proposed algorithm can easily be extended to embed trees in hexagonal as well as other types of processor arrays. © 1991 Academic Press, Inc.

1. INTRODUCTION

Tree machines have been shown to be suitable for solving a large class of problems like sorting, searching, finding cliques in graphs, graph coloring, and the general class of divide-and-conquer algorithms. Horowitz and Zorat [5] have observed that the binary tree is an important interconnection network and have addressed several issues related to the use of a binary tree in parallel computations.

Advances in VLSI technology over the last decade have made the design and implementation of a tree machine on a single large-area IC, or a small number of such ICs, possible. Each IC contains a large number of processing elements and interconnection links. As a result, the probability that one or more of these elements will either have a manufacturing defect or become inoperable is not insignificant. To increase the yield and reliability of these tree machines and to allow their use in the presence of faulty processors, two different approaches have been proposed.

In the first approach, a tree with redundant processors and links is implemented in such a way that spare processors can be incorporated into the tree architecture in the event of a manufacturing defect or fault. Such an approach toward a fault-tolerant tree machine was proposed by Raghavendra *et al.* [10]. Their scheme provides a spare node for each level in the binary tree, with redundant links to ensure protection against failures. Hassan and Agarwal [4] proposed a modular fault-tolerant tree. In their scheme, a fault-tolerant module capable of tolerating a single fault is designed. Many copies of the basic module when properly interconnected yield a complete fault-tolerant binary tree. A drawback of the above schemes is that because the switching circuitry must not be too complex, each redundant processor has only a small set of processors that it can replace. Consequently, as is often the case, if there is a cluster of defects [13], it is possible to run out of processors in a localized region while having an ample supply of redundant processors elsewhere.

Another scheme which is better suited to handle clusters of defects was recently proposed by Howells and Agarwal [6]. In their scheme, spare processors, along with an elaborate setup of switches and redundant links, are placed in quasioptimal points in the layout to achieve the largest possible yield enhancement. The amount of redundant hardware (processors, switches, and links) is optimized with respect to a fault model and not with respect to the number of faults actually present in individual arrays. The actual number of faults will vary from array to array. As a result, it is possible that too much redundancy will be provided for some arrays with a smaller-than-estimated number of defects, or too little for other arrays with a greater-than-estimated number of defects.

In the second approach, called the *embedding approach* to the incorporation of defect and fault tolerance into tree machines, a tree is embedded in a "host" architecture such as a square or hexagonal array of processors. Whenever a processor in the host architecture becomes faulty, the embedding is repeated to route around the faulty processors. The embedding is accomplished by using some of the functioning nodes as *connecting elements* (CEs) which merely act as a conduit to pass messages between *processing elements* (PEs).

An important advantage of the embedding approach over the first approach is that the host architecture is a general-purpose array which is dynamically configured to match the needs of the currently used algorithm. The configuration can be changed as the algorithm changes. This flexibility is not available in the first approach.

* Supported in part by NSF under Grant MIP-8805586 and by ONR under Grant N00014-87-K-0796.

0743-7315/91 $3.00
Copyright © 1991 by Academic Press, Inc.
All rights of reproduction in any form reserved.

A second advantage of the embedding approach is the ease with which fault clusters can be allowed for. The scope of redundancy in this approach is automatically global. By contrast, elaborate structures of redundant links and switches are required to allow for global sparing in the first approach.

Several embedding algorithms (e.g., [2, 3, 7–9, 11, 12, 14]) have already been proposed in the literature, the earliest being the H-tree embedding [9]. Most of these algorithms only deal with *fault-free* arrays. The Diogenes approach [11] is a scheme for embedding trees in a *linear* array with faulty elements. However, the problem of embedding binary trees in *rectangular* or *hexagonal* processor arrays in the presence of random faults is known to be NP-complete, and hence no efficient deterministic algorithm exists.

A different approach to the embedding of tree machines was proposed by Koren and Pomeranz [8] and is applicable to partially faulty rectangular and hexagonal arrays. The main idea behind this approach can be stated as follows: when a subtree cannot be mapped in its original position because of faulty elements, it is "moved" in a predetermined direction, until it either can be mapped successfully or cannot be moved any more. This approach also uses CEs to route around faulty processing elements.

In this paper, we present an algorithm which embeds binary trees in rectangular arrays *in the presence of faulty processors*. The essence of the algorithm lies in its random and distributed nature. Unlike the algorithm in [8], our algorithm has no predetermined pattern for embedding the tree. Instead, it attempts to embed a given subtree in the currently available region in a completely random manner. A node which is specified as a root of a k-level tree chooses two neighboring nodes, say α and β, at random, and the algorithm continues recursively: nodes α and β consider themselves as roots of a $(k-1)$-level tree. Subtrees are grown in parallel, and the process continues until the leaves of the tree are reached. One major characteristic of the algorithm is its ability to obtain embeddings in the face of fault clusters in the processor array.

The main criterion used to evaluate the quality of the embedded tree is the *maximum root-to-leaf* (MRL) distance. The MRL distance is considered an important parameter to determine how fast a given computation will run on a tree machine.

A second measure is the time required to find an acceptable embedding and is motivated as follows. Clearly, when a random algorithm is employed, we cannot expect any single run to necessarily achieve a satisfactory (small MRL) embedding. Consequently, the user should run the algorithm a few times until a satisfactory embedding is obtained. The overall expected time taken by the algorithm to obtain an acceptable embedding is therefore an important measure of the quality of the algorithm. This quantity can be computed from the probability distribution of the MRL values produced by the algorithm.

Another measure that would be of interest is the probability that this algorithm will find an embedding on a partially faulty array, given that at least one embedding is theoretically possible. Unfortunately, this measure would be impractical to obtain. As we have already pointed out, the embedding problem is NP-hard. Given a partially faulty array, it is a nonpolynomial problem to determine whether an embedding is possible on that array. Consequently, it would not be practical to determine this measure.

It should be pointed out that the algorithm does *not* attempt to minimize the area of embedding. The region of the array over which the embedding is to take place is specified to the algorithm, which then proceeds to use as much of it as it finds necessary. For this reason, the *processor utilization*, i.e., how many of the available processors are used, is not of much interest since such a measure would have no practical bearing on the performance of the algorithm, apart from that which is already taken into account by the maximum root-to-leaf measure.

The rest of the paper is organized as follows. In Section 2, we present a detailed description of the algorithm. Sections 3 and 4 evaluate the algorithm's performance when the fault distributions are uniform and clustered, respectively. In Section 5 we demonstrate a method for further reducing the mean execution time. We conclude in Section 6 with an overall evaluation of the algorithm.

2. DESCRIPTION OF THE ALGORITHM

The algorithm described below is capable of embedding binary trees in processor arrays which contain faulty processing elements. A few definitions for understanding the working of the algorithm are given.

The following types of node are recognized by the algorithm.

Faulty Node. A node α appears to be faulty to a neighboring node β if either the processing element located in α is faulty or the communication link connecting α and β is faulty. Hence a node α might appear to be faulty from β but fault-free from a third node γ.

Fault-Free and Busy. A fault-free and busy node is one which has already been allocated to a node in the tree. It might be operating as

- a processing element capable of performing computations, or
- a connecting element whose only function is to pass messages between PEs.

Fault-Free and Nonbusy. A fault-free and nonbusy node is one which is operational and has not yet been allocated to any node in the tree.

Child, parent. A node α is said to be the *child* of the *parent* β if both α and β are processing (not connecting) elements,

and α represents a child of β in the tree. Contrast this with the definitions of *predecessor* and *successor,* defined below.

Successor, predecessor. A node α is said to be the successor of (its predecessor node) β if either α is a child of β or α is used as a connecting element which is part of the path between β and one of its children.

There are five different types of messages passed between the nodes. They are described below:

Type 1. A message from node α to a nonbusy and fault-free neighboring node allocating it to the root of a $(k-1)$-level subtree and requesting it to embed this subtree.

Type 2. A message from a node α to its predecessor indicating that it is unable to embed the subtree rooted at α. This is a negative acknowledgement message.

Type 3. A message from node α to its predecessor indicating that it has successfully completed the embedding of the subtree rooted at α. This is a positive acknowledgement message.

Type 4. A message from node α to its previously selected successor requesting it to deallocate itself and all the nodes it had allocated.

Type 5. A message from a node (which has previously received a deallocation request) to its predecessor indicating that the deallocation has been completed.

The input to the algorithm is a number k which signifies the level of the complete binary tree which is to be embedded onto the rectangular array.

The output from the algorithm, if successful, is an embedding of a k-level complete binary tree on the processor array. The algorithm declares failure if it is unable to find an embedding.

Outline of the Algorithm

A formal description of the algorithm in pseudocode is presented in the Appendix. Below, we present an informal description. Prior to the execution of the algorithm, the processor array is initialized. This involves setting all operational PEs in the rectangular array as nonbusy. An operational PE α near the center of the array is then chosen to act as the root of the binary tree to be embedded. Next, a path of operational PEs from this PE to the edge of the array is found and all PEs along this path are marked as busy. These PEs then serve as connecting elements which enable the root of the binary tree to communicate with the external world. A simple message passing algorithm, where every PE (starting with α) sends a message to each of its neighboring PEs, can be used. The algorithm terminates as soon as a PE β on the edge of the array receives a message. Each PE (starting with β) then identifies the sender of its message and the entire path from β to the root α is retraced. All PEs along this path from α to β are marked as connecting elements. The array is now ready to run the algorithm.

Let α be the node which has been allocated to be the root of a k-level binary tree in the initialization phase of the algorithm. Node α starts by selecting at random two fault-free and nonbusy neighbors, say β and γ, out of its three neighbors, the fourth being α's predecessor, i.e., either its parent or a CE on a path leading to its parent. Node α then orders the two selected neighbors β and γ (by sending them a type 1 message each) to embed in parallel a $(k-1)$-level binary tree. Each of these two neighbors will return either a positive acknowledgement (type 3 message) if the embedding for their respective subtree was completed successfully or a negative acknowledgement (type 2 message) if all attempts to embed the subtree have failed.

If either one of the two selected neighbors β or γ returns a negative acknowledgement, α then transmits a deallocate message (message type 4) to both β and γ. Nodes β and γ send similar deallocate messages to their successors and hence the entire subtrees rooted at β and γ are deallocated. When β and γ have deallocated their subtrees, they send messages (message type 5) to α indicating that they have done so. Node α then starts over by once again selecting two random neighbors out of its three available neighbors. The maximum number of times α attempts to select a set of two neighbors is one of the parameters to the algorithm and is called the `PE retry count`.

If, after a number of repetitions determined by the `PE retry count`, α is still unable to find a set of two neighbors, it selects a single fault-free and nonbusy neighbor (say β) and declares itself to be a CE. β now starts recursively the same procedure to which α was assigned. In the case where α has only a single fault-free and nonbusy neighbor (two faulty or busy neighbors), the above step of choosing a single neighbor would be performed right in the beginning.

Node α, now in the role of a CE, awaits an acknowledgement from β. If β returns a negative acknowledgement, it sends β a deallocate message, waits for a confirmation of deallocation, and proceeds to select a new neighbor at random. The number of times a single neighbor is selected is the other parameter to the algorithm and is called the `CE retry count`. If the number of attempts by α exceeds the `CE retry count`, then a negative acknowledgement is sent by α to its parent.

The case when α receives a positive acknowledgement is handled in one of two ways. If α is operating as a PE, it must receive two positive acknowledgements before it sends a positive acknowledgement to its parent. If α is operating as a CE, the receipt of a single positive acknowledgement prompts it to send a positive acknowledgement to its predecessor.

The deallocate message assumes precedence over all other message types described above. The receipt of a deallocate message by a node α prompts it to deallocate itself and all its successors, regardless of whether α has received zero, one, or two positive or negative acknowledgements prior to receiving the deallocate message.

A copy of the algorithm executes at each of the operational nodes during the embedding process. Each node essentially idles until it receives an incoming message from one of its neighboring PEs. A message contains the message type, the sender's identification, and a level number indicating the level of the subtree. In response to a message, a node changes its internal state and might, if necessary, send messages to its neighbors. After processing a message, a node once again goes back to its idle state.

Figures 1 and 2 show two example embeddings, obtained by the above algorithm, of a 7-level complete binary tree on a partially faulty processor array. The faulty nodes are marked by circles in these figures. Each node in these two arrays was independently determined to be either faulty with a given probability p or fault-free with probability $1 - p$. Figure 1 shows a processor array of size 18×18 and node fault probability $p = 0.15$. Figure 2 shows a processor array with array size 20×20 and node fault probability $p = 0.2$. Figure 3 depicts the embedding of an 8-level complete binary tree in a 30×30 array with a node fault probability of $p = 0.05$.

To summarize, our algorithm has the following characteristics:

(i) it is recursive in nature,
(ii) it is nondeterministic, and
(iii) it is distributed.

We should stress that this algorithm is specifically meant to embed trees in partially faulty arrays. It is not meant to be used when the array is fault-free: there are better, and deterministic, ways of doing that, for example, an H-tree embedding. For this reason, direct comparisons of this algorithm with deterministic algorithms designed for fault-free arrays are inappropriate.

The nondeterministic nature of the algorithm presents many problems in its analysis. The use of standard analytical tools, such as Markov modeling, is impractical for this case since the action of any processor in choosing its children affects the availability of children for the neighboring processors, which in turn tends to affect the choices open to the

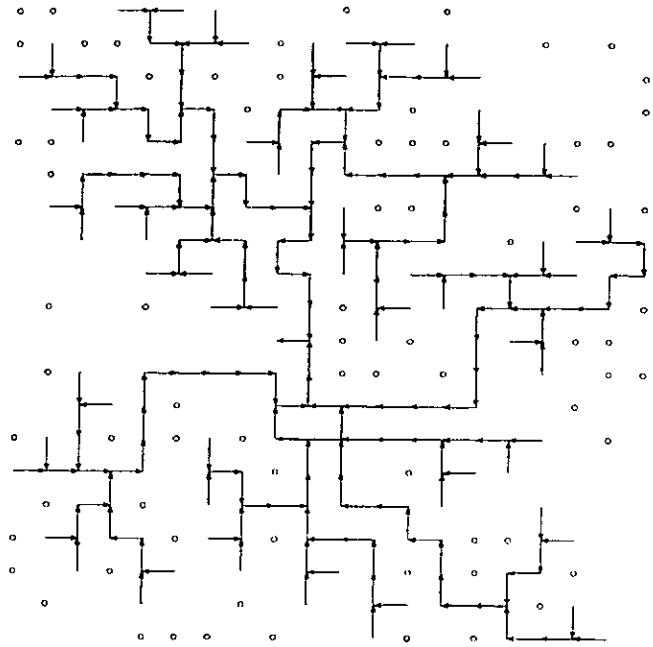

FIG. 2. A 7-level binary tree in a 20×20 array ($p = 0.2$).

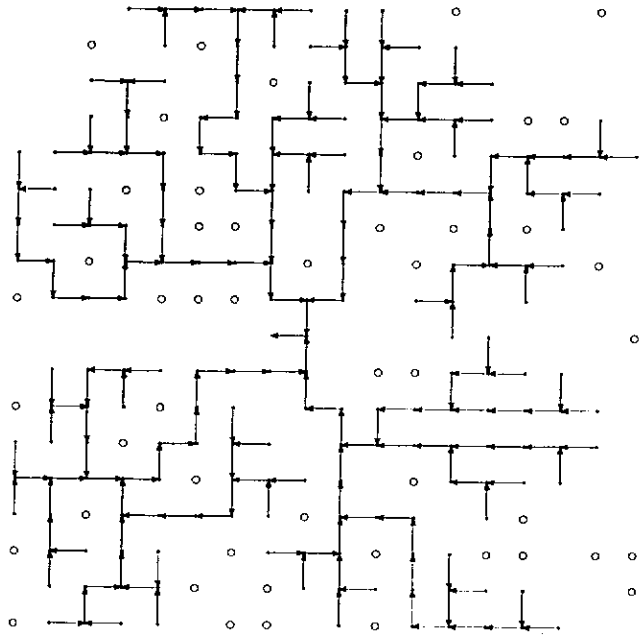

FIG. 1. A 7-level binary tree in an 18×18 array ($p = 0.15$).

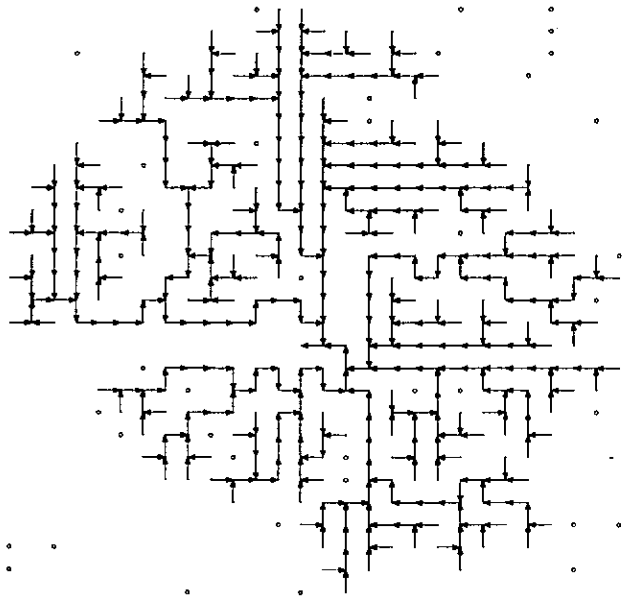

FIG. 3. An 8-level binary tree in a 30×30 array ($p = 0.05$).

neighbors of these neighboring nodes, etc. The result is a state space of a large number of dimensions, which cannot be decoupled from one another. This leads to a number of computations which grows exponentially with the size of the tree being embedded.

Therefore, extensive simulation runs were conducted in order to study the performance of the algorithm. The results of these simulations are presented in Sections 3 and 4.

3. ALGORITHM PERFORMANCE: UNIFORMLY DISTRIBUTED FAULTS

The first group of simulation experiments was conducted with the intent of determining the algorithm's ability to embed binary trees in a grid with uniformly distributed faults. The main purpose was to determine how the MRL distance varies with an increase in failure probability and array size.

The experiments were conducted as follows. In a given array, each node was independently determined to be faulty with failure probability p. The algorithm was then executed and an embedding obtained. For each value of failure probability and array size, 500 runs were made. The MRL distance was averaged over the 500 runs and their results tabulated. Figure 4 shows the average of the MRL distance over 500 runs for a 127-node tree embedded in arrays of different sizes.

As can be seen in Fig. 4, the MRL distance tends to increase as the node failure probability increases. This is understandable because the algorithm has to use a larger number of connecting elements to route around the faulty nodes. However, the other dependence—that of the MRL distance on the array size—is not as monotonic. As the array size increases, the tree has more room to grow, and so one might expect the MRL distance to increase. However, as the array size decreases, simulation results show that in a large number of instances there is often a long train of connecting elements along the edge of the array which contribute to a larger MRL distance. The reason for a long train of connecting elements along the edge is obvious—the algorithm, prevented from growing outward, attempts to find an empty region within the array to grow its subtree. This region is more likely to exist along the edges, as areas toward the interior of the array would have already been used for nodes which are part of the upper levels of the tree.

The next step in evaluating the performance of our random algorithm was to determine how long one had to execute it in order to obtain an acceptable embedding. Also, as this algorithm is specifically designed to embed trees in the presence of faulty nodes, it was necessary to determine how the mean execution time varies with an increase in node failure probability. The grid size was also used as a parameter to study its effect on the mean execution time.

This set of experiments was conducted as follows. For each value of node failure probability and array size, a fault

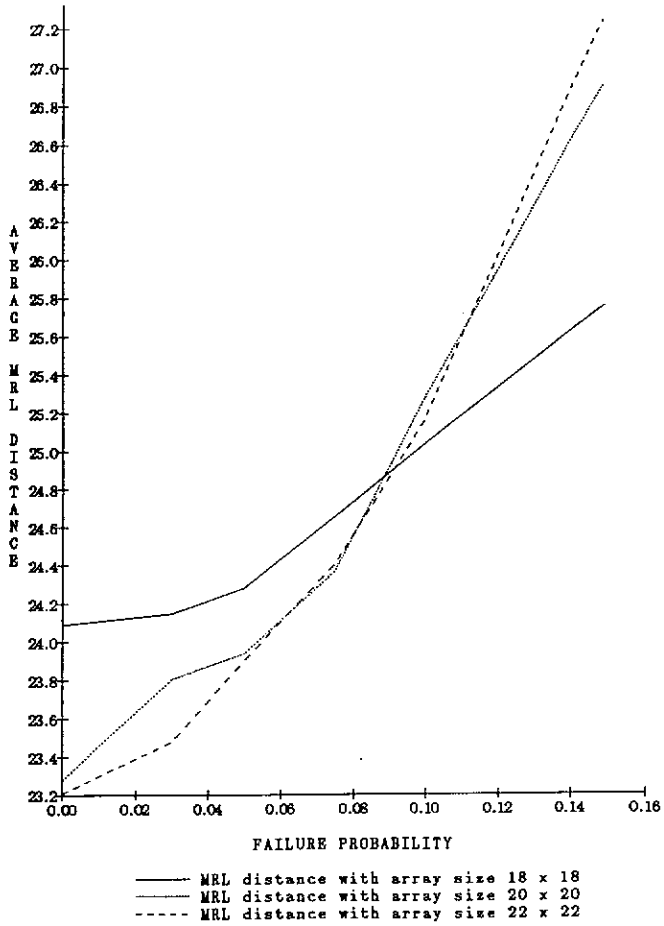

FIG. 4. Average MRL distance as a function of the failure probability p.

pattern was chosen at random. The algorithm was then executed to obtain 500 embeddings for this fault pattern. The resulting mean execution time for a 7-level tree is depicted in Fig. 5. The execution time is measured in units of *single PE operations*. A single PE operation is defined as the number of machine cycles (typically about 50) required for a PE to process completely any incoming message.

The dependence of the mean execution time on the node failure probability and the array size is as one might expect. As the array size increases, the algorithm has more room in which to embed the tree and so tends to successfully complete the embedding in a shorter time. However, as the failure probability increases, the algorithm finds potential avenues for growth increasingly blocked by failed nodes. For this reason, a successful completion of the tree embedding takes a larger number of attempts and therefore more time.

Since this is a *random* algorithm, the first embedding it comes up with is not necessarily a good one, and several runs might be necessary to find one. In Fig. 6, we present a simulation-based estimate of the complementary probability distribution function (CPDF) of the MRL distance. Call this function $P_1(x, p)$. That is,

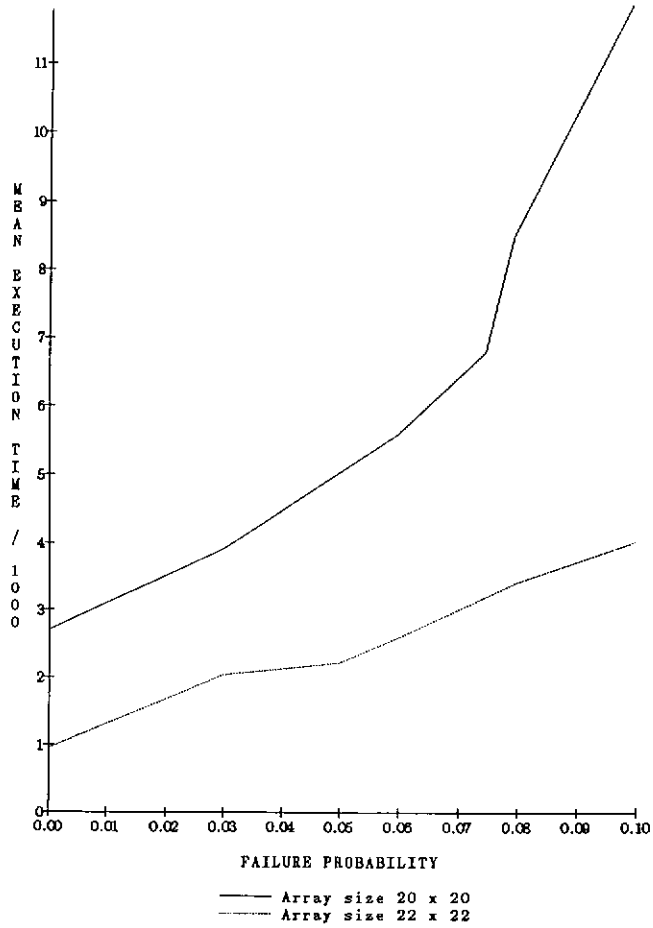

FIG. 5. Mean execution time as a function of the failure probability p for a 7-level tree.

$$P_1(x, p) = \text{Prob}\{\text{MRL distance} > x \text{ when failure probability} = p\}.$$

If we run the embedding algorithm n times and choose that with minimum MRL distance, the CPDF of this minimum quantity is given by

$$P_n(x, p) = [P_1(x, p)]^n.$$

We can easily get a feel for the actual time in seconds required to find an embedding. Suppose we wish to determine how many runs, $n_{run}(x, p, \beta)$, need to be made before we obtain, with probability β, an embedding whose MRL distance is no greater than x. The embedding is assumed to be in an array whose failure probability is p. Clearly,

$$n_{run}(x, p, \beta) = \min\{n \mid P_n(x, p) < (1 - \beta)\}.$$

From Fig. 6, for a 22×22 array, with $x = 16$ and $p = 0.03$, we have $n_{run}(x, p, 0.99) = 90$. Assuming a clock rate of 10 MHz, 50 machine cycles per PE operation, and from Fig. 5, a mean execution time of $2 \cdot 10^3$, we get an embedding time of

$$\frac{2 \times 10^3 \times 50 \times 90}{10^7},$$

which is 0.9 s.

4. ALGORITHM PERFORMANCE: CLUSTERED FAULTS

Increasing the level of integration to larger chips with more transistors causes the defect distribution in VLSI to deviate significantly from a uniform distribution. An extensive study of this has been carried out by Stapper [13], who showed that the effect of defect clustering is modeled more closely by a negative binomial distribution. The performance of the algorithm in the presence of clustered faults is thus important

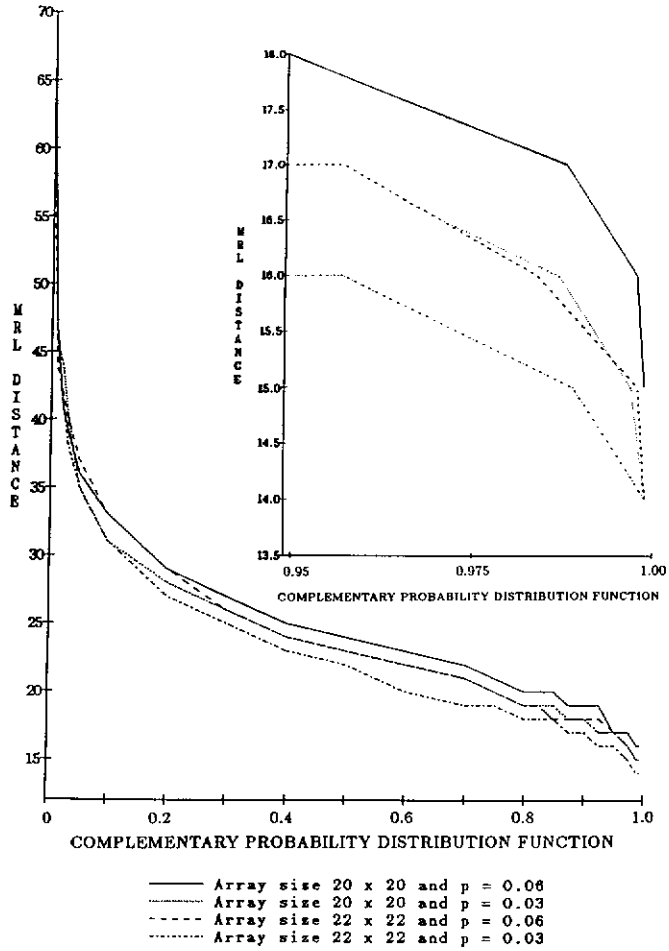

FIG. 6. The complementary probability distribution function of the MRL distance. (Inset shows detail.)

in this context. Specifically, it is important to study what effect the degree of clustering has on the MRL distance.

Prior to the running of the algorithm and the generation of an embedding, faulty nodes were distributed throughout the array according to a negative binomial distribution. The entire array was first divided into quadrants, each containing 25 processing elements. The negative binomial distribution was used to determine the number of faults in each quadrant. These faults were then uniformly distributed within each quadrant. The negative binomial distribution used is

$$P(X = k) = \frac{\Gamma(\bar{\alpha} + k)}{k!\Gamma(\bar{\alpha})} \frac{(AD/\bar{\alpha})^k}{(1 + AD/\bar{\alpha})^{k+\bar{\alpha}}},$$

where X is a random variable designating the number of faults within the quadrant with area A and defect density D. Each processor in the array was assumed to occupy unit area, and hence D (the number of defects per unit area) is equal to the fault probability p of Section 3. The clustering parameter is $\bar{\alpha}$ and must be greater than zero. The smaller the value of $\bar{\alpha}$, the more intense the clustering.

For each value of $\bar{\alpha}$, 500 runs were conducted and the average of the MRL distance was calculated as before. Figure 7 shows the results of the simulation runs conducted with various values of $\bar{\alpha}$. Values of $\bar{\alpha}$ above 2.5 correspond to a near-uniform distribution and hence were not studied. Figure 7 indicates that the MRL distance is essentially independent of $\bar{\alpha}$.

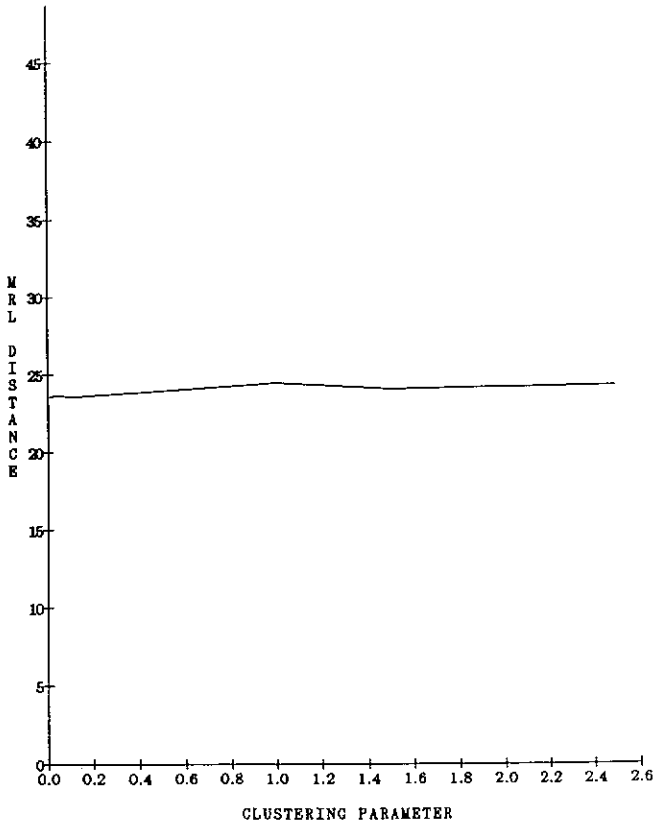

FIG. 7. Average MRL distance as a function of the clustering parameter $\bar{\alpha}$.

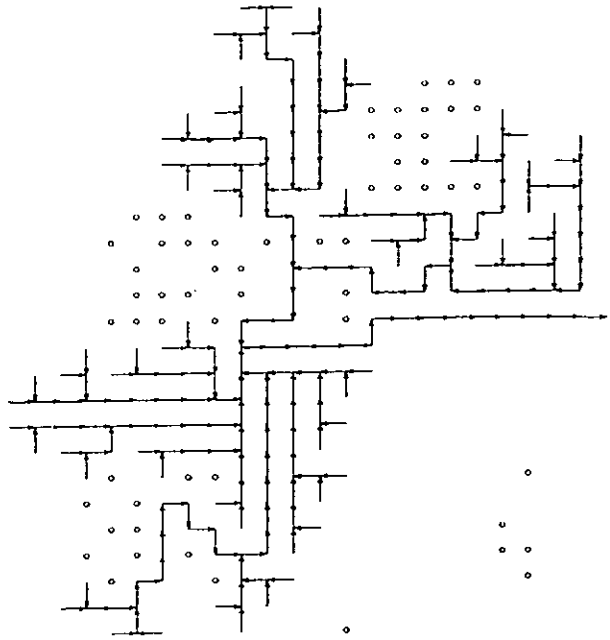

FIG. 8. A 7-level binary tree embedded in an array with clustered defects ($\bar{\alpha} = 0.1$, $D = 0.2$).

Figures 8 and 9 show two embeddings obtained with $\bar{\alpha}$ equal to 0.1 and 2.5, respectively. Figure 8 is an example of an array where the defects are highly clustered. As can be readily seen, the algorithm successfully routes around the clusters of defects to obtain an embedding with MRL distance equal to 26. The defect density (D) in this example is equal to 0.2.

Figure 9 shows an embedding in an array where the effect of defect clustering is less pronounced. The defect density is equal to 0.1 and the resulting embedding has MRL distance equal to 15.

5. CONCURRENT RUNS

A closer look at the design of the algorithm of Section 2 reveals that the algorithm underutilizes the resources of the processor array during the embedding process.

Any nonleaf node, α, is at the root of a k-level tree for some $k > 0$. It starts by requesting two of its free neighbors to be the roots of $(k-1)$-level trees and then idles until it receives a positive or negative acknowledgment from them. If k is large, then a considerable time will most likely elapse before node α receives such an acknowledgment. As a result, the node will be idle for a large fraction of the embedding time.

A similar argument can be put forward for leaf nodes.

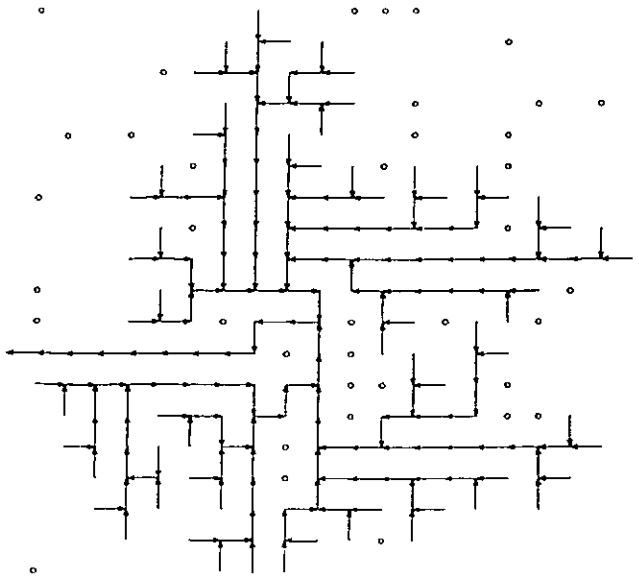

FIG. 9. A 7-level binary tree embedded in an array with clustered defects ($\bar{\alpha} = 2.5$, $D = 0.1$).

Once allocated as a leaf, a node remains in the idle state, unless it receives a deallocate message from its parent. In fact, the maximum utilization of the processors in the array occurs when all the leaf nodes of the binary tree are activated by an incoming message. The number of leaf nodes constitutes only half the total number of nodes in a binary tree and only a small fraction of the total operational nodes available in the processor array.

It is therefore apparent that this scheme greatly underutilizes processors in the array. To avoid such wastage of processing capacity, we adopt the simple stratagem of growing more than one tree in parallel. The number of trees grown in parallel is referred to as the *degree of concurrency*.

In the modified implementation of the algorithm, any node in the processor array can simultaneously participate in up to m different trees being grown in parallel, where m is the degree of concurrency. All data structures in the original algorithm are replicated m times to store the necessary status information for the m trees. A tree number is included in all messages to identify the tree to which the message pertains.

Each node then operates as follows. Upon receipt of a message, the node examines the tree number, accesses and updates status information for that tree, and sends messages to its neighbors with the tree number field appropriately set. If more than one message is received, the messages are processed on a first-come-first-serve basis by the node.

An immediate question with this scheme is, What should the value of m be for any given tree size? If the value of m is too small, there will still be a number of idle processors in the array at any given instant. However, if the value of m is made too large, there will be a long queue of messages at every node, which would require large local storage at each node. This is the only significant factor to consider when choosing m: since local storage is all that is being used, there is little, if any, context-switching overhead.

Simulation runs for the concurrency scheme were made and their results are shown in Fig. 10. For each value of m, the elapsed time (in PE operation units) required to generate 500 embeddings was recorded. These values were divided by the time required to generate 500 embeddings with $m = 1$ and the ratio was plotted in Fig. 10. In each case the array size was kept constant. Runs were made for tree sizes of 15, 31, 63, and 127 nodes.

It can be seen that as the size of the tree is increased, the value of m needs to be larger. From Fig. 10, for a tree size of 15 nodes and $m = 6$, it would take approximately one-fifth the time to generate 500 embeddings that it would take if the degree of concurrency were equal to one. For a tree size of 127 nodes and $m = 20$, it would take approximately one-twentieth the time it would take if the degree of concurrency were equal to one. Finally, the curves show that beyond a certain value of m, increasing the degree of concurrency does not improve the throughput in terms of the number of embeddings generated per unit time. Quasioptimum values of m for each tree size are indicated by circles on the curves of Fig. 10.

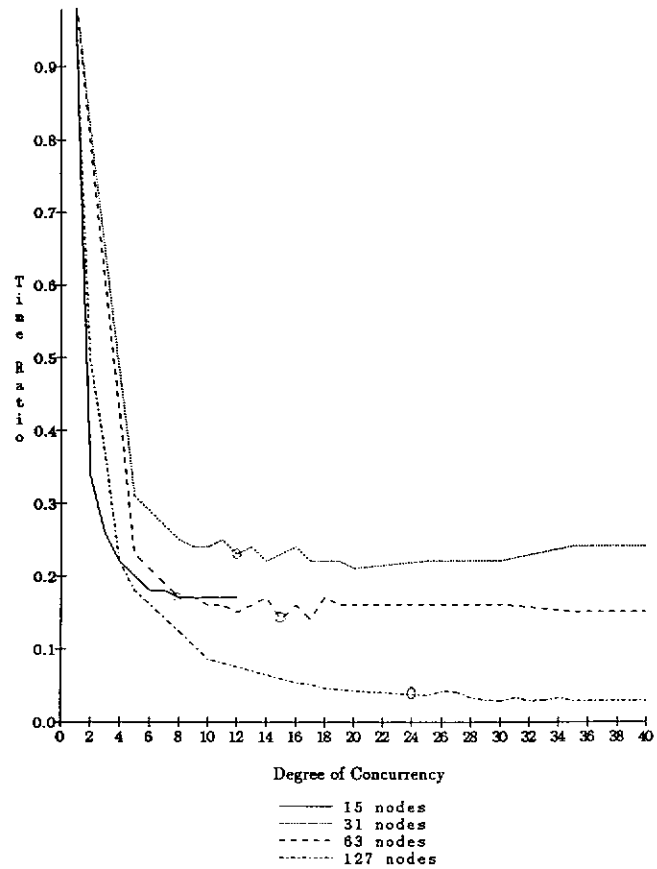

FIG. 10. Execution time ratio as a function of the degree of concurrency.

6. CONCLUSIONS

An overall evaluation of the proposed algorithm to embed binary trees in rectangular arrays produces the following conclusions:

Advantages

(i) The main advantage of our embedding algorithm is its ability to handle a random distribution of faulty nodes and links. This is borne out by the simulation results using random fault patterns.

(ii) The algorithm is capable of handling fault clustering, which is inevitable in VLSI. The simulation results indicate that the degree of clustering has no substantial impact on the performance of the algorithm. This is a definite advantage compared to other schemes which require high levels of redundancy to accommodate fault clustering.

(iii) The algorithm is also capable of embedding *incomplete* binary trees in fault-free as well as partially faulty arrays. Figure 11 shows an example of a 7-level incomplete binary tree containing 95 nodes embedded in a partially faulty array with fault probability $p = 0.1$. The resulting MRL distance is 14.

(iv) The parallel nature of the algorithm has the added advantage that one can use the processor array itself to carry out the embedding, rather than doing so off-line. The algorithm, due to its simplicity, does not require the use of any special instructions at the hardware level. Moreover, the memory requirement at each node is fairly low.

(v) Finally, the algorithm can be easily extended to embed binary trees in arrays of higher degree, as well as trees of greater degree than binary. Thus, this algorithm could be easily extended to run, for example, on hexagonal arrays or a hypercube architecture.

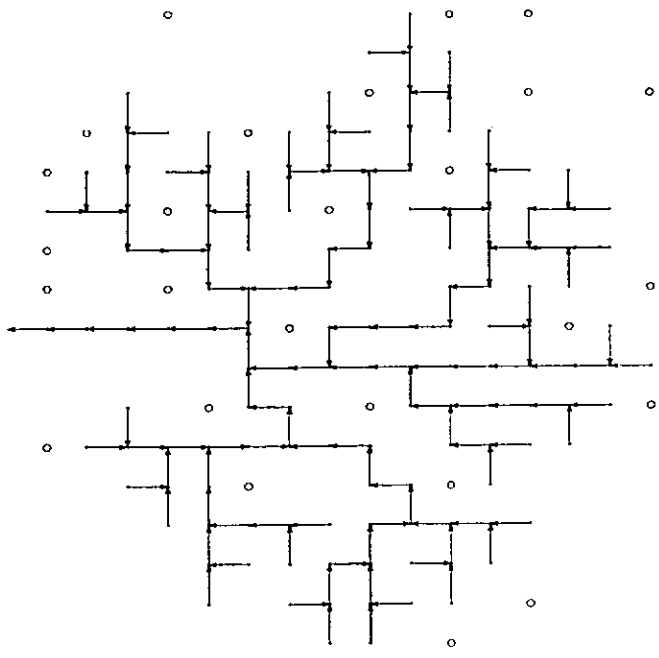

FIG. 11. A 7-level incomplete binary tree with 95 nodes.

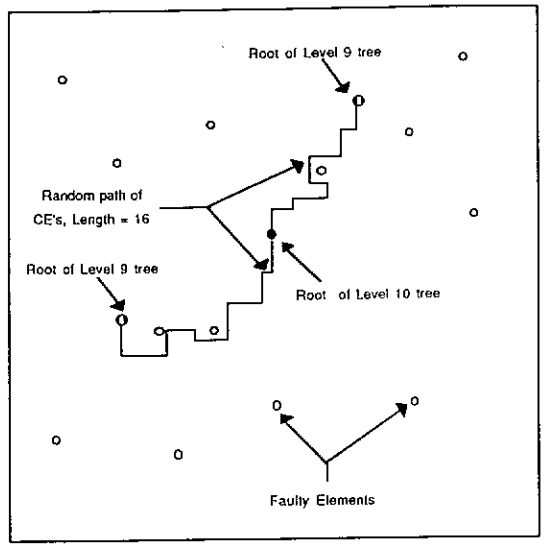

FIG. 12. A modified algorithm for embedding a 10-level binary tree.

Disadvantages

The main disadvantage of the proposed algorithm is the increased execution time needed to generate embeddings for trees where the number of nodes is very large, since as the number of nodes increases, the potential for interference between the "growth" of the various subtrees increases also. So, for large trees (say, for trees of more than l_{max} levels for some appropriate l_{max}), additional measures must be taken. l_{max} is not a constant, but rather it depends on the algorithm's execution time that is considered acceptable in a particular implementation.

This problem can be solved by introducing certain heuristics within the algorithm. It is apparent that the increased execution time is a result of the collisions which occur at the upper levels of the tree because of the large number of nodes which need to be embedded at the lower levels. One possible solution is to separate adjacent nodes at the upper levels with a long chain of connecting elements, thereby reducing the probability that adjacent subtrees will compete with each other for the same region of the processor array. The length of the chain of connecting elements which separate a node at level k from a node at level $(k-1)$, where k is greater than l_{max}, can be the same as that for the H-tree type of embedding. This value grows exponentially [7] and should provide sufficient room for subtrees to grow without excessive collisions. For embedding subtrees of level $l_{max} - 1$ and less, the algorithm can revert to its original form.

In our implementation of the algorithm we have decided

to set $l_{max} = 9$. Thus, a tree of 10 or more levels can benefit from the incorporation of the above modification into the algorithm. Figure 12 shows how the modified algorithm works to embed a 10-level binary tree. The PE which is the root of the 10-level tree grows two paths of CEs in random directions. The length of each path is equal to 16, which is the same as that for a 10-level H-tree embedding. The two PEs at the end of each path then grow 9-level subtrees using the original form of the algorithm.

APPENDIX: FORMAL DESCRIPTION OF THE ALGORITHM

A *free neighbor* indicates a nonbusy, fault-free, and accessible neighboring node. A neighbor of α is accessible if the communication link connecting it to α is fault-free. It should be noted that a node which is inaccessible from α may still be accessible from its other neighbors and can therefore be allocated to another tree node.

```
switch ( message type ){
  case 1 : if (leaf node)
             set my_node type to PE;
             send message type 3 to parent;
           else
           if (two free neighbors are
           available)
              set my_node type to PE;
              decrement the level number by
              one;
              send message type 1 to two free
              neighbors selected at random;
           else
           if (only one free neighbor is
           available)
              set my_node type to CE;
              send message type 1 to that node
              with the same level number;
           else {* no free neighbor is
           available *}
              send message type 2 to parent;
  case 2 : if (node type is PE)
              send message type 4 to both sons;
           else {* node type is CE *}
              send message type 4 to son;
  case 3 : if (node type is PE)
              if (two messages of type 3 have
              been received)
                 send message type 3 to parent;
           else {* node type is CE *}
              send message type 3 to parent;
  case 4 : if (node type is PE)
              if (node is not a leaf)
                 send message type 4 to both
                 sons;
              else
                 send message type 5 to parent;
                 deallocate my_node;
           else {* node type is CE *}
              send message type 4 to son;
  case 5 : if (node type is PE)
              if (two messages of type 5 have
              been received)
                 if (message type 4 received
                 earlier)
                       send message type 5 to
                       parent;
                       deallocate my_node;
                 else {* the node originating
                 the deallocate
                         message was reached *}
                       if (PE retry count not
                       exceeded)
                          increment PE retry
                          count;
                          send message type 1 to
                          two free neighbors
                          selected at random;
                       else
                          set my_node type to CE
                          choose a free neighbor
                          at random;
                          send message type 1 to
                          that node;
           else {* node type is CE *}
              if (message type 4 received
              earlier)
                       send message type 5 to
                       parent;
                       deallocate my_node;
              else {* the node originating the
              deallocate
                       message was reached *}
                 if (CE retry count not
                 exceeded)
                       increment CE retry count;
                       choose a free neighbor at
                       random;
                       send message type 1 to
                       that node;
                 else
                       send message type 2 to
                       parent;
                       deallocate my_node;
```

REFERENCES

1. Fussel, D., and Varman, P. Fault-tolerant water scale architecture for VLSI. *Proc. 9th Annual Symposium Computer Architecture,* Apr. 1982.
2. Gordon, D., Koren, I., and Silberman, G. M. Embedding tree structures in VLSI hexagonal arrays. *IEEE Trans. Comput.* **C-33,** 1 (Jan. 1984), 104–107.
3. Gordon, D. Efficient embeddings of binary trees in VLSI arrays. *IEEE Trans. Comput.* **C-36,** 9 (Sept. 1987), 1009–1018.
4. Hassan, A. S., and Agarwal, V. K. A modular approach to fault-tolerant binary tree architectures. *IEEE Trans. Comput.* **C-35** (Apr. 1986), 356–361.
5. Horowitz, E., and Zorat, A. The binary tree as an interconnection network: Applications to multiprocessor systems and VLSI. *IEEE Trans. Comput.* **C-30,** 4 (Apr. 1981), 247–253.
6. Howells, M. C., and Agarwal, V. K. A cluster-proof scheme for improving yield and reliability of large area binary trees. In W. R. Moore, W. Maly, and A. J. Strojwas (Eds.). *Yield Modeling and Defect Tolerance in VLSI.* Adam Hilger, Bristol, UK, 1988, pp. 175–182.
7. Koren, I. A reconfigurable and fault-tolerant VLSI multiprocessor array. *Proc. 8th Annual Symposium Computer Architecture,* June 1981, pp. 425–442.
8. Koren, I., and Pomeranz, I. Distributed structuring of processor arrays in the presence of faulty processors. In W. R. Moore, A. McCabe, and R. Urquhart (Eds.). *Systolic Arrays.* Adam Hilger, Bristol, UK, 1987, pp. 239–248.
9. Mead, C., and Conway, L. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980.
10. Raghavendra, C. S., Avĩzienis, A., and Ercegovac, M. Fault tolerance in binary tree architectures. *IEEE Trans. Comput.* **C-33,** 6 (June 1984), 568–572.
11. Rosenberg, A. L. The diogenes approach to testable fault tolerant arrays of processors. *IEEE Trans. Comput.* **C-32,** 10 (Oct. 1983), 902–910.
12. Snyder, L. Introduction to the configurable, highly parallel computer. *IEEE Comput.* **15,** 1 (Jan. 1982), 47–56.
13. Stapper, C. H. On yield, fault distribution, and clustering of particles. *IBM J. Res. Develop.* **30,** 3 (May 1986), 326–338.
14. Youn, H. Y., and Singh, A. D. Near optimal embedding of binary tree architecture in VLSI. *Proc. 1988 International Conference on Parallel Processing.* IEEE Computer Society, Silver Spring, MD, June 1988.

Received June 13, 1989; accepted May 10, 1990

DIPAK SITARAM holds a B.S. in electronics and communication engineering from the Birla Institute of Technology, India, and an M.S. in electrical engineering from the University of Massachusetts, Amherst. He is currently working for IBM Corp. in East Fishkill, New York, in the area of VLSI physical design.

ISRAEL KOREN received the B.Sc., M.Sc., and D.Sc. degrees from the Technion–Israel Institute of Technology, Haifa, in 1967, 1970, and 1975, respectively, all in electrical engineering. He is currently a professor of electrical and computer engineering at the University of Massachusetts, Amherst. Previously he was with the Departments of Electrical Engineering and Computer Science at Technion. He has been a consultant to Digital Equipment Corp., National Semiconductor, Tolerant Systems, and ELTA–Electronics Industries. Dr. Koren's current research interests are fault-tolerant VLSI and WSI architectures, models for yield and performance, floor-planning of VLSI chips, and computer arithmetic. He edited and coauthored the book *Defect and Fault-Tolerance in VLSI Systems* (Vol. 1, Plenum, New York 1989). He was also a Co-Guest Editor for *IEEE Transactions on Computers,* special issue on High Yield VLSI Systems, April 1989.

C. M. KRISHNA received the B.Tech. degree from the Indian Institute of Technology, Delhi, the M.S. degree from Rensselaer Polytechnic Institute, Troy, New York, and the Ph.D. degree from the University of Michigan, Ann Arbor, all in electrical engineering. Since September 1984 he has been on the faculty of the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. He was a visiting scientist at the IBM T. J. Watson Research Center during the summer of 1986. His research interests include reliability modeling, queueing and scheduling theory, and distributed architectures and operating systems.