Complex Systems **1** (1987) 81–106

Self-Configuration of Defective Cellular Arrays

Myoung Sung Lee
Gideon Frieder
Departments of Electrical Engineering and Computer Science,
University of Michigan,
Ann Arbor, MI 48109, USA

Abstract. Rapid advances in VLSI technology are making it feasible to consider the construction of parallel computers with very large numbers of cells. One promising architecture for such computers is a VLSI cellular array that interconnects many simple processing cells on a single large chip or wafer. It will be very difficult, however, to make a chip of this kind without many defects. With a fixed interconnection pattern between cells, the whole cellular array may not be usable when there are any defects. Furthermore, an architecture with a fixed interconnection pattern is limited in the range of computations that can support efficiently. By providing reconfiguration mechanisms, a VLSI cellular array can be designed so that it can be reconfigured for fault-tolerance and specialized for various computations.

This paper discusses a *massively fault-tolerant cellular array*, which contains identical cells with connections only to immediate neighbors, where the cells and the connections may be defective with a high probability. The cell can function as a processing element, as a memory, or as a switching element that connects to other cells. On the defective array, a large cluster of interconnected working cells is formed, and the working cells in the cluster are configured into a graph that determines the function of the array.

The detailed architecture of the massively fault-tolerant cellular array is described, and the distributed algorithms for forming the cluster of working cells and configuring the cells into a linear array, a two-dimensional array, a binary tree, and signal flow graphs for various filters are presented. Simulation data are presented when both cells and connections are defective with various probabilities.

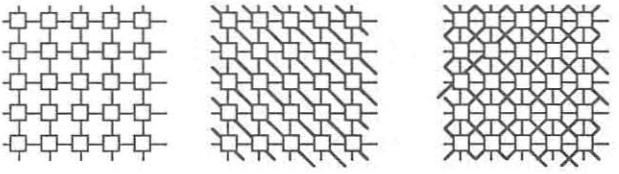
1. Introduction

Since cellular automata provide a theoretical model for a parallel computer, many proposals and attempts to build cellular computers have been made.

Cellular array machines built in the past are SIMD (single instruction multiple data) machines with a small number of cells like Solomon [18] and ILLIAC [15,2]. More recent cellular array machines are typically SIMD machines with a large number of simple bit-processors with storage at each cell. Examples include CLIP [5], a 96 by 96 array with 32 bits of storage at each cell; Massively Parallel Processor [3], a 128 by 128 array with 1K bits per cell; the Distributed Array Processor [11], a 64 by 64 array with 4k bits per cell; and the Adaptive Array Processor [12], whose building block is a single chip 8 by 8 array with 96 bits per cell. Because of the limitations and high cost/performance ratio of available components, however, these machines were quite limited in power. With the rapid progress in integrated circuit technology, low-cost, high-density, fast VLSI devices are making it feasible to consider the construction of large-scale MIMD (multiple instruction multiple data) computers that were previously considered too complicated. One promising area in this class of machines is a VLSI processor array that interconnects a very large number of processing cells on a large single chip or wafer.

For large-scale multiprocessors, however, VLSI technology imposes a local communication constraint, since communication in VLSI medium is very expensive in terms of area, power and time consumption [16]. This local communication constraint makes cellular array machines particularly attractive because of their simple connection patterns. On any interconnection pattern, however, only a rather small set of computations can be performed efficiently. Furthermore, when defects appear on the chip, the whole cellular array may become unusable. By using the cells as switches for changing the interconnection pattern, ia cellular array can however be designed so that it can be reconfigured for fault-tolerance and for specialization to specific computations. By changing the connection pattern to suit the specific computation, one multiprocessor system can support a wide range of computations efficiently, and by changing the connection pattern to route around the defects, the cellular array can function in the presence of the many defects that are expected on a large chip.

To take full advantage of reconfigurable cellular arrays, it is desirable to arrange for distributed self-configuration, in which cells determine the configuration pattern without knowledge of the state of the array as a whole. Efficient procedures of this kind have not, however, been given for the case of many defective cells [14,1,10,7]. Nevertheless, as shown in this paper, distributed self-configuration can be done efficiently when there is sufficient computing power in each cell. In this paper we will describe a cellular array machine that can be configured efficiently despite many defects. The cellular array, here called the *massively fault-tolerant cellular array*, is an array of identical cells with connections only to immediate neighbors where each cell can function as a processing element, as a memory, or as a switching element that connects to other cells. Input and output terminals are connected only at the boundaries of the array. The cellular array anticipates the occurrence of massive defects in the cells and in the inter-



(a) square array (b) hexagonal array (c) octal array

Figure 1: Interconnection patterns of the cellular array.

connections. The cellular array is designed in such a way that there exists a set of working cells which maintains a desired processing capability despite many defective cells and defective interconnections, by changing the connection pattern.

To maintain processing capability despite many defective cells and interconnections, cells in the massively fault-tolerant cellular array need to have some mechanism to identify defective cells and interconnections, and configure themselves around these defective elements. By employing built-in self-testing techniques, we should be able to devise a testing mechanism by introducing additional testing hardware. In this work, we have assumed that by some mechanism, each cell knows if its neighbor and the connection to its neighbor is working or is defective.

After the defective cells are identified, a big cluster of interconnected working cells is formed from a subset of all working cells in the array. The cells without an adequate number of working neighbors are pruned out from the cluster. Then cells in the cluster are then configured into a graph that implements the function of the array.

In the following sections, we will describe the architecture of the massively fault-tolerant cellular array, and the algorithms for the formation of the cluster, the pruning of the cluster, and the configuration of the cells into a linear array, a two-dimensional mesh, and a complete binary tree. Simulation of the massively fault-tolerant array and simulation results of the formation of the cluster, pruning, and configuration of cells into a linear array, a two-dimension mesh, and a complete binary tree are shown on arrays of size 40×40, 80×80 and 120×120.

2. Massively fault-tolerant cellular array

The three regular interconnection patterns studied in this paper are shown in figure 1. They are selected to study the effect of the number of the neighbors on the behavior of the cellular array. The three arrays with the three interconnection patterns of figure 1 are called *square array, hexagonal array,*

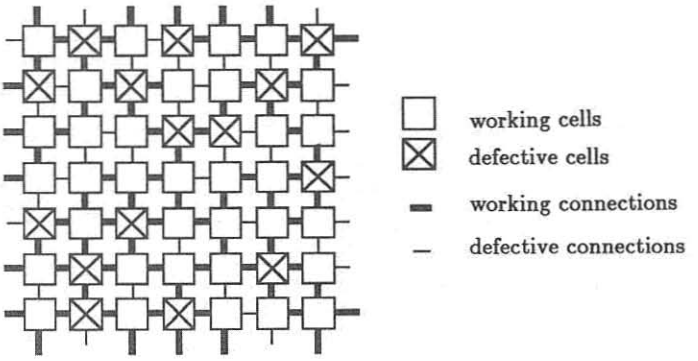
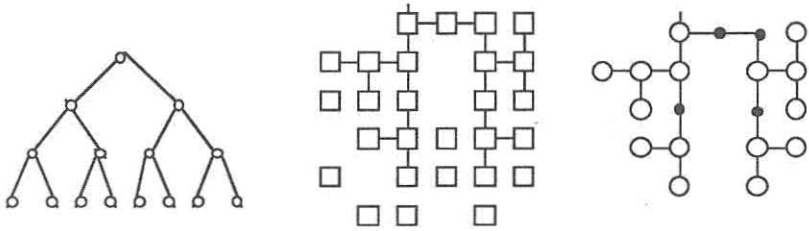


Figure 2: A defective square cellular array.

and *octal array*, respectively. Figure 2 shows a square array with defective cells and defective interconnections. Note that although the initial array is regular, the ensuing array is not (see figure 2), as the faults cause breakdown in the regularity of the array. Though there are many defects, there are still many working cells in the array so that useful computation can be performed. The computations that the array is intended to perform will determine how the working cells are configured. The logical interconnection of cells for any particular computation can be represented by a graph called the *computation graph* in this paper. The configuration of cells into a computation graph is the process of embedding the computation graph in the defective array. The embedding of the computation graph in the defective array is represented by a graph, called the *connection graph*. For example, the tree in figure 3(a) is a computation graph; figure 3(b) shows an embedding of the tree on the defective square array; figure 3(c) is the connection graph of the embedding of figure 3(b). In figure 3(b), some cells are mapped to the nodes of the computation graph, while other cells are used to connect the cells that are mapped into the nodes of the graph. The cells mapped into the nodes of the graph are called the *computation cells*, and the cells used to connect the computation cells are called the *connection cells*. The connection cells are represented by dots on the connection graph.

A cell in the massively fault-tolerant cellular array is configured into a computation graph by identifying the logical connections of the cell. For example, a cell is configured into a binary tree by saving in special registers the directions of the neighbors as a father, a left child, and a right child of the cell. The configuration can be changed by changing the contents of the registers. The configuration is performed in a distributed fashion: each cell makes a decision only with the information that is available at the cell.



(a) computation graph (binary tree)

(b) embedding of a tree on a defective array

(c) connection graph of the embedding(b)

Figure 3: A computation graph, its embedding, and a connection graph.

2.1 Architecture of a Cell

Each cell has a processor, local memory, and a set of registers used for self-configuration. The processor is an instruction set processor with a small instruction set. The instructions include usual arithmetic, logical, data transfer, program control operations, and send and receive for input and output. For the communication with neighbors, the cell has Communication Registers, and for the self-configuration, the cell has an ID Register, Neighbor Status Registers, and Configuration Registers. The Configuration Registers consist of a Status Register, a Pattern Register, and Connection Registers. Figure 4 shows the architecture of a cell in the square array. On the hexagonal and octal array, the number of Communication Registers and Neighbor Status Registers increase to six and eight, respectively. The purposes of these registers are as follows:

ID Register: Stores the row and column indices of the cell in the array.

Neighbor Status Register, NS[0..N-1 :] Stores the status of the neighbors. It shows whether communication to each neighbor is working or defective. When either the neighbor is defective or the connection to the neighbor is defective, the NS shows that communication to that neighbor is defective. Here N is the number of immediate neighbors.

Status Register, SR: Stores the current status of the cell. The states of a cell are "idle", "live", "pruned", "computation", and "connection". The status of all working cells are initially "idle". The clustering procedure changes the status of the cells belonging to a big interconnected cluster to "live", and the pruning procedure changes the status of the working cells without an adequate number of cells to "prune", so that they are not used in the configuration procedure. The configuration procedure changes the status of the "live" cells to

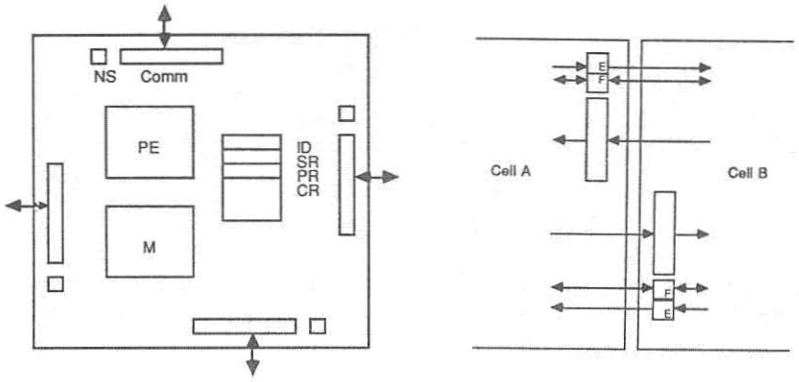


Figure 4: Architecture of a cell and communication register of the cellular array.

"computation" or "connection", depending upon whether the cell is used as a computation cell or a connection cell.

Pattern Register, PR: Stores the current configuration pattern. The configuration patterns that we currently recognize are linear array, two-dimensional mesh, complete binary tree, spanning tree, and FIR and IIR filter graphs.

Connection Register, CR[0..N-1 :] Stores the directions of neighbors in the configuration specified by the Pattern Register, PR. The meanings of the CR[0..N-1] are different for each configuration: when PR specifies that the configuration is a linear array, CR[0] is the direction of the predecessor, and CR[1] is the direction of the successor; When PR specifies a binary tree, CR[0], CR[1], and CR[2] are the directions of the father, the left child, and the right child; When PR specifies a two-dimensional mesh, CR[0], CR[1], CR[2], CR[3] are the directions of the up, right, down, and left neighbors of the cell. When PR specifies a spanning tree, CR[0] is the direction of the father, and CR[1..N-1] are the directions of the sons, etc.

2.2 Communication mechanism of the array

Cells communicate with other cells by sending and receiving messages through Communication Registers. Each Communication Register provides a one-way communication between two neighbors, and between a boundary of two cells, two Communication Registers provide two-way communication. Figure 4 shows a pair of Communication Registers. The arrows in figure 4 show the directions of the data. The fields of the Communication Register are as follows:

Full Bit, FB: Shows if the Communication Register is full or empty.

Enable Bit, EB: Shows if the other cell is willing to receive the message. If EB is 0, the other cell may not read the Communication Register. EB is used to prevent deadlock.

Messages, Mesg: Contains the messages. It includes a tag bit that shows if the message is a datum or an instruction.

The FB is used to synchronize the communication between two cells. A cell can read a Communication Register when the Communication Register is full, and after the cell reads the Communication Register, FB is reset to 0. If a cell wants to read a message from an empty Communication Register, the cell waits until the Communication Register is full again. A cell can write to an empty Communication Register, and if a cell initiates write to a full Communication Register, the cell has to wait until the Communication Register is empty. This provides the synchronization between two cells.

At the processor level, communication is handled by "send" and "receive" instructions. "Send (direction, message)" writes the "message" on the Communication Register located at "direction": Return from "Send" acknowledges that the message is written to the Communication Register. "Receive (direction)" reads a message from the Communication Register in "direction", and "Receive (any)" reads a message from any Communication Register that is full: Return from "Receive" acknowledges that a message has been read from the Communication Register in "direction."

The Communication Register and "Send", and "Receive" instructions provide the asynchronous communication protocol required in the data flow computation model. These are also used for input and output with the outside.

2.3 Operation of the array

In the massively fault-tolerant cellular array, an external machine is attached to the boundary cells to control the operation of the array. The external machine initiates the operations of the array, and provides the data to the array by sending messages to the array. The external machine will be called a *Controller*.

When the massively fault-tolerant cellular array is powered on, Neighbor Status Registers, NS[0..N-1], of each cell are set to indicate the status of the communication to neighbors of each cell (working or defective) by some testing mechanism. The hardware and algorithms for the testing have not been studied yet. We assumed that the testing can be done by some mechanism.

After the Neighbor Status Registers are set correctly, the controller initiates the clustering procedure which identifies the largest cluster of interconnected working cells in the array. If the size of that cluster is adequate, the cells in the cluster are given identification numbers that are stored in

the ID Registers of all cells in the cluster. After all the cells in the cluster are given identification numbers, the controller initiates the pruning procedure which prunes out the cells in the cluster without an adequate number of working neighbors. The controller then initiates the configuration procedure to configure the cells in the cluster into a desired computation graph. When the Controller sends a message to a cell at the boundary of the array, the cell relays the message to its neighbor cell, and the message will propagate through the working cells. The cells will be configured into a graph specified on the message. The configuration of the cells is finished when all the cells set their Configuration Registers correctly and the boundary cell connected to the controller returns a message to the controller.

3. Formation of the cluster of cells

Since the cells in the cellular array connect only to the immediate neighbors, when a cell wants to communicate with another cell that is not directly connected to it, the message should be relayed by intervening working cells to the target cell. Therefore, a small cluster of working cells surrounded by defective cells cannot be used. The array can be useful only when a large cluster of of interconnected working cells is formed in the array.

We can use the *percolation theory* [17,6] to predict if a large cluster appears on the array, and if the cluster appears, to predict the size of the cluster. According to the percolation theory there exists a *critical probability* such that when the probability that a cell is working is more than the critical probability, there appears an infinite cluster of interconnected working cells in the defective infinite array of cells.

3.1 Percolation theory and the cellular array

Consider a lattice L defined as a graph of N sites (or vertices) and M bonds (or lines). In most cases of practical interest, L will be a regular two or three dimensional lattice of finite or infinite extent. In the *bond problem*, each bond of L is *occupied* (or *open*, or *working*, etc.) with probability p or *vacant* (or *blocked*, or *defective*, etc.) with probability $1-p$. Occupied bonds are *connected* if they meet at a common site, and a connected set of s bonds form a *bond cluster* of size s. In the *site problem*, each site is occupied with probability p and vacant with probability $1-p$. Occupied sites are connected to form *site clusters* if they are adjacent through the bonds of L.

For an infinite lattice L there is a critical probability $p_c = p_c(b, L)$ or $p_c(s, L)$ for the bond or site problems such that for $p < p_c$ all clusters will be finite while for $p > p_c$ there will, with positive probability, be an infinite cluster in L. The infinite cluster is called a *percolation cluster*. We can define the *percolation probability*, $P(p)$, as the probability that a site, chosen at random, belongs to an infinite cluster. One defines the *critical*

probability, p_c, as

$$p_c = \sup\{p|P(p) = 0\}.$$

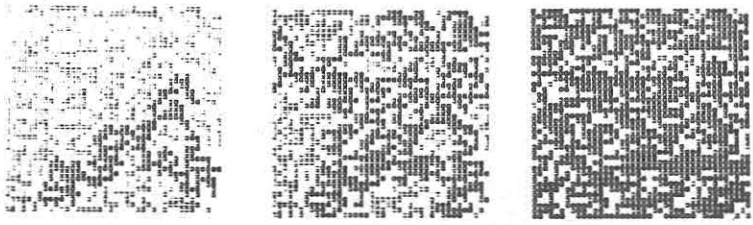
The sites in the percolation model corresponds to the cells of the massively fault-tolerant cellular array, and the bonds in the percolation model corresponds to the connections between cells. In the site percolation problem, all the bonds are assumed to be occupied and only the sites can be vacant; this corresponds to the assumption that all connections are working and only cells can be defective. In the bond percolation problem, all the sites are assumed to be occupied and only the bonds can be vacant; this corresponds to the assumption that all cells are working and only connections can be defective.

Since the area of a cell is much bigger than the area of a connection in the cellular array, and the defect probability of an integrated circuit is at least proportional to the area of the integrated circuit, the defect probability of a cell is much greater than that of a connection. Therefore, as a first approximation, the array can be modeled by the site percolation model. On the site percolation model of the defective array, we can take into account that connections can be defective by associating connections with neighboring cells. When a connection is defective, the cells associated with the defective connection are considered defective. When we need to use the working cells connected to the defective connections, the defective array should be modeled by the combination of site and bond percolation problem [9].

In percolation theory the *percolation cluster* is an infinite cluster that appears on the infinite lattice. On the cellular array of finite size, we define the *percolation cluster* as the largest cluster that is connected to all four borders of the array when the array is rectangular. Using the percolation theory, we can predict that the percolation cluster will appear in the defective cellular array when the probability that a cell is working, p, is more than the critical probability, p_c of percolation theory.

Figure 5 shows the formation of the clusters in a square array where the critical probability is 0.59. Here cells in solid boxes belong to the largest cluster. When p is 0.5, the percolation cluster does not appear (figure 5(a)); when p is 0.61, a thin percolation cluster appears (figure 5(b)); when p is 0.75, a thick percolation cluster appears (figure 5(c)).

However, as we can see in figure 5(b), even though the percolation cluster appears in the array, when p is not high enough, the percolation cluster is "thin", contains many branches, and does not include many boundary cells. The cells on the thin percolation cluster may not be used effectively because communication between two cells in the thin cluster is difficult, and configuration of cells into a computation graph is not efficient. Furthermore, input and output on the thin percolation cluster is difficult due to the lack of the boundary cells. Therefore the thin percolation cluster in figure 5(b) may not be useful. When p is high enough, the percolation cluster that appears in the array is "thick", and has many boundary cells.



(a) p = 0.5
a percolation cluster
does not appear

(b) p = 0.61
a thin percolation
cluster appears

(c) p = 0.75
a thick percolation
cluster appears

Figure 5: The formation of clusters on the square array.

The cells in the thick cluster may be used effectively for configuration and computation. The cluster in this case is shown in figure 5(c).

The usefulness of the percolation cluster depends on the computation graph that will be embedded on the percolation cluster. For example, a percolation cluster may be considered adequate enough to embed a linear array but the same percolation cluster may not be adequate to embed a two-dimensional array.

3.2 Formation of the cluster

When p is more than p_c, we can form a percolation cluster of working cells in the defective array. The cluster is formed by connecting the working cells into a spanning tree which spans all the working cells connected to a certain boundary cell. The controller sends a message to a working cell at the boundary, where the message specifies that a spanning tree of working cells be formed. The cell that received the message from the controller becomes the root of the spanning tree.

After a spanning tree is formed with the cell that received a message from the controller as the root of the spanning tree, the cell returns the number of cells connected into a spanning tree to the controller. If the number is greater than the critical number, the cells in the spanning tree are taken as the percolation cluster. The critical number of cells is chosen as

$$\frac{\text{critical probability} \times N^2}{2} \tag{3.1}$$

where N^2 is the size of the array. Because of the distribution of size of the clusters, a cluster of less than this size cannot be the percolation cluster. If the number is less than critical number, another message is sent to some

other working cell on the boundary, and a new spanning tree is formed with the new cell as the root of the spanning tree. This process continues until a percolation cluster is found, or the controller gives up finding a percolation cluster in the defective array.

Figure 6 shows the algorithm for connecting the cells into a spanning tree. Here, scopes of 'if', 'then', 'else', 'repeat', etc. are determined by the indentation, and comments are enclosed by '{' and '}'. The first two lines show the contents of the message. A line beginning with 'Receive' shows the content of the message when a cell receives a message, and a line beginning with 'Return' shows the content of the message when a cell returns a message. The message field includes a tag that tells if the message is an instruction or data. Each cell receives a message with an instruction that says 'configure into a spanning tree', and returns a message with a data that is the number of cells in the subtree. All the cells execute the same program.

When a cell receives a message from a neighbor, the cell changes its state by setting the Status Register, SR to "live", and it saves the direction of the neighbor on CR[0] as the father of the cell. The cell then sends the message to its working neighbors. If a neighbor returns the message that the neighbor is a part of the spanning tree, the direction of the neighbor is saved on CR[1..N-1] as a son. The spanning tree grows in depth first order. The enabling and disabling of communication registers is necessary to prevent deadlock. Figure 7 show the percentage of working cells that are in the largest cluster in the 120 by 120 square, hexagonal, and octal array, respectively. The experimental results are generally in agreement with percolation theory. When connections are not defective, we can see that more than 90% of working cells belong to the percolation cluster when p is more than 0.7 on the square array, when p is more than 0.6 on the hexagonal array, and when p is more than 0.5 on the octal array. As the connections become defective, these numbers decrease as shown in figure 7. The cellular array can be used on the plateau region of figure 7 the area of which increases as the degree of the array increases.

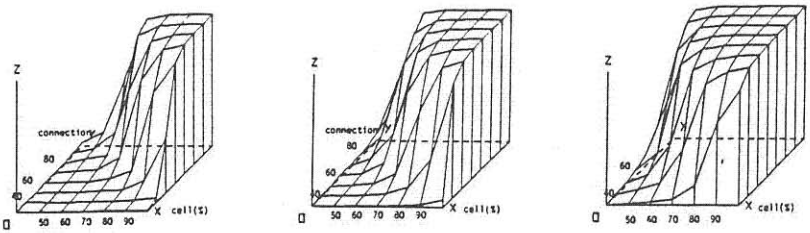
3.3 Assignment of identification numbers

After the working cells are connected into a spanning tree, the cells on the spanning tree are assigned identification numbers. The identification number of a cell is the row and column indices of the cell in the array. The identification number is saved in the ID register.

The controller sends a message to the root cell of the spanning tree where the first field of the message specifies the operation, and the next fields are the row and column indices of the root cell. When a cell receives the message, the cell saves the row and column indices on the Id register, and computes the Id of the son. Then the cell sends the message with the computed Id to the son. When the cell receives the message from the son, it iterates the same operations on the next son.

```
Receive: mesg1: opcode telling the cell to configure into a spanning tree
Return:  mesg1: number of cells in the subtree of the cell

L: recv(any, mesg1)
   if the cell is not idle_cell
   then send(dir, 0) {message came here already}
        goto L
   else {I am a idle cell}
        status := live_cell
        number of sons := 1 {including this cell}
        connect the sender as my father

        repeat for all neighbors
            send to a working neighbor with enabled communication
            disable all communication except the neighbor
            recv from the neighbor
            enable all communication
            if mesg1 > 0 then connect the cell as my son.
                                 number of sons := number of sons + mesg1

        send(father,number of sons)
        goto L
```

Figure 6: Clustering algorithm.



(a) square array (b) hexagonal array (c) octal array

Figure 7: Percentage of working cells that are in the largest cluster.

3.4 Pruning

After a percolation cluster of working cells is formed in the defective array, the cells in the percolation cluster can be configured into a computation graph. However, some cells in the percolation cluster form a single-width, dead-end branch, and they may not be configured effectively. Furthermore, they may slow down communications between cells. To facilitate the configuration of working cells and the communication between cells, we may prune out the dead-end branch of cells from the percolation cluster.

Pruning of the dead-end branches from the cluster can be generalized to *pruning-to-k*. The Pruning-to-k operation prunes out the cells which are connected to less than or equal to k working neighbors. Here k is called the *level* of the pruning. Pruning of the dead-end branch corresponds to pruning-to-1: the cells connected to only one working neighbor are pruned out from the cluster. Pruning is applied repetitively until no more cells are pruned.

By pruning the cells from the cluster, we can have a cluster of tightly connected cells. After the pruning-to-k operation, all the cells in the cluster are connected to at least $k+1$ working neighbors. This can facilitate the configuration of cells into a graph and the communication among the cells in the cluster. When the working probability is adequate, most of the cells in the percolation cluster are connected to several working neighbors [13].

4. Configuration of cells

The cells have to be configured into a general computation graph which specifies the function of the array. Before the configuration of the cells into a general computation graph, we studied the configurations into three particular graphs: linear array, complete binary tree, and two-dimensional array (mesh). Many computations can be done efficiently on these graphs.

The configuration of cells into a computation graph is the process of embedding the computation graph on the defective array. Since the cells that do not belong to the percolation cluster cannot be used, the computation graph is embedded on the percolation cluster. Note that when the percolation cluster appears on the defective array, most of the working cells belong to the percolation cluster.

The efficiency of the configuration into a graph G, e_G, is defined as

$$e_G = \frac{\text{number of cells used as computation cells}}{\text{number of working cells in the cluster}} \times 100 \qquad (4.1)$$

The delay $d_G(C_1, C_2)$ between the two cells, C_1 and C_2, in a configuration into a graph G is defined as one plus the number of connection cells between the two cells C_1, and C_2. Therefore, the delay between two directly connected cells is 1, and the delay is 2 when there is one connection cell between two cells. The *maximum delay* of the configuration is the maximum delay among all two adjacent computation cells, and the average

delay of the configuration is the average of all delays among two adjacent computation cells.

We define the *degree* of a graph G, d_G, as the average degree of the vertices of the graph. To configure the cells in the defective array into a computation graph of degree d_G efficiently, the number of neighbors on the array, or the degree of the array, d_A, needs to be greater than d_G. When d_A is less than d_G, the efficiency of the configuration becomes low. We can expect that cells can be configured into a linear array ($d_G = 2$), and a tree ($d_G = 3$) efficiently on the square array ($d_A = 4$), the hexagonal array ($d_A = 6$), and octal array ($d_A = 8$). But the configuration of cells into a mesh ($d_G = 4$) on the defective square array may not be as efficient as the configuration on the hexagonal array or on the octal array.

In the following sections an overview of configuration procedures into a linear array, a tree, and a mesh are described. The detailed algorithms can be found in [13].

4.1 Linear Array

In the linear array, every cell has two neighbors: the predecessor, and the successor. The configuration of cells into a linear array is the process of identifying predecessors and successors and saving the directions of the predecessors and successors in the Connection Registers, CR[0], and CR[1].

The configuration procedure consists of three parts: Linear, Extend, and Join. Procedure Linear grows the linear array into the defective array of cells. When the linear array is grown in the defective array by the Procedure Linear, Procedure Extend finds the cells which are not in the linear array, but which can be connected into the linear array. Then Procedure Join connect the cells identified by Procedure Extend into the linear array. By combining the three procedures, Linear, Extend, and Join, most of the cells in the cluster are connected into the linear array.

The controller initiates the configuration by sending a messages to a cell at the boundary of the array. The message consists of the fields specifying the operation, the number of cells to be connected into the linear array, and the direction of the successor neighbor. When a cell B receives a message from a neighboring cell A, the cell B sets the Pattern Register PR to "linear array", and saves the direction of the cell A on CR[0] as a predecessor. Then the cell B tries to grow the array by adding a neighbor as its successor. First, if the neighbor C specified as the successor on the message is working, the message is sent to C. If C is connected to the linear array, the linear array grows from C again. The cell C returns the message to B with the number of cells on the linear array after C. Then cell B saves the direction of the cell C on CR[1] as a successor, and increases the number of cells by one, and returns the message to the predecessor, A. If C fails to be connected into the linear array, then the neighbor on the direction of the growth of the linear array as specified on the message is tried. If this fails too, then any working neighbor is tried. If all fail, the linear array retracts

to the cell A, and growth of linear array is tried at cell A again. Since the cells do not know the global state of the network, the linear array can be grown into the dead-end, and the cells may have to backtrack often.

When Procedure Linear is finished, the cell at the boundary which received the message from the Controller returns the number of cells connected into the linear array to the controller. If the number of cells is less than the number the Controller wants, the controller sends a new message to the cell. The new message consists of the fields specifying Procedure Extend and the number of cells to be joined to the linear array. The cells which were not part of the linear array but adjacent to the linear array are identified and joined into the linear array.

With the three configuration procedures, Linear, Extend, and Join, most of the cells are configured into a linear array when the working probability of a cell is adequate. Figure 8 shows the algorithm of Procedure Linear. Figure 9 shows the cells connected into a linear array on the defective array. Figure 10 shows the percentage of cells in the cluster that are connected into a linear array on the square array, on the hexagonal array, and on the octal array.

From figure 10, we can see that most working cells are connected into a linear array with adequate working probabilities of cells and connections. Since degree of the linear array is two, the configuration of cells into a linear array should be efficient on all arrays even when working probability of a cell is not high. As the degree of the array increases, efficiency of the configuration increases rapidly. When the connections are not defective, on the square array, when the working probability of cells is 80%, more than 85% of the working cells are connected into the linear array. On the octal array, with the working probability of cells 60%, about 90% of the working cells are connected into the linear array.

Since all computation cells are connected directly to the other computation cells without intervening connection cells, no delay has been introduced, and the average delay is 1.

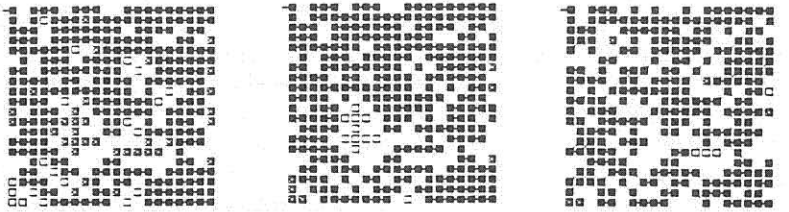
4.2 Tree

In the complete binary tree, every cell has three neighbors: a father, a left child, and a right child. The working cells of the defective array are configured into a complete binary tree by setting their Connection Registers CR[0..2] to the directions of a father, a left child, and a right child of each cell respectively, and saving the level of the cell in the tree on CR[3]. (The level of the leaf node is defined as one, and the level of a father is one more than that of its child.) Some working cells are used as the nodes of the tree, which are *computation cells*, and some are used as *connection cells*, which are used to connect computation cells.

Since the topology of the binary tree and that of the array of cells do not match, we need to use many working cells as connection cells even when there are no defects in the network. Koren [10] studied embedding of a tree

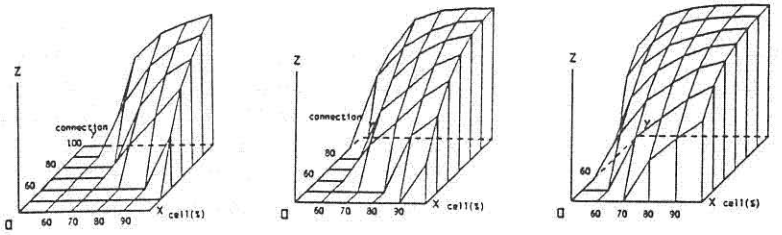
```
Receive: mesg1: opcode telling the cell to connect into a linear array (op_line)
         mesg2: direction of the neighbor to be tried first
         mesg3: direction of the linear array
         mesg4: number of cells to be connected into the linear array
Return:  mesg1: number of cells connected into the linear array
         mesg2: maximum number of cells connected into the linear array

L: recv(any, mesg1, mesg2, mesg3, mesg4)
   if the cell is not live_cell
   then send(dir, 0, 0)
   else { connect me as part of the linear array }
        predecessor := dir of the neighbor which sent the message
        make a table of neighbors to try to connect
        repeat for each neighbor in the table
            if a neighbor is working and communication is enabled then
                send(neighbor, op_line, mesg2, mesg3, mesg4)
                disable all communication registers except the neighbor
                recv(neighbor, mesg1, mesg2)
                enable all communication registers
                if mesg2 > 0 then
                    connect the neighbor into a linear array
                    successor := dir of the neighbor
                    goto L

        send to predecessor
        goto L
```

Figure 8: Linear array configuration algorithm.



(a) square array, p = 0.8 (b) hexagonal array, p = 0.8 (c) octal array, p = 0.7

Figure 9: Configuration of cells into linear arrays on the defective array.



(a) square array (b) hexagonal array (c) octal array

Figure 10: Percentage of cluster cells that are connected into a linear array.

in a defective square array, but his procedure allows few defects, and its efficiency is very low when there are many defects. The algorithm sets all working cells in the row and the column of the defective cell as connection cells, thereby making a reduced array without defect of one less row and one less column for each defects. Note that when there are many defects, none of the rows and columns will be without defective cells.

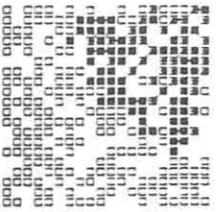
The algorithm we devised allows efficient configuration even when many cells are defective. The algorithm has two parts: Tree and Retract. Procedure Tree connects the cell into a tree, and Procedure Retract retracts the subtree when it cannot increase the level of a subtree.

The Controller initiates the configuration into a tree by sending a message to a cell at the boundary of the array. The message consists of a field specifying the operation and the level of the tree desired. If the cells are successfully configured into a tree of the level specified on the message, the cell returns the level of the tree. The Controller then increases the level of the tree by one, and sends the message again to the cell until the desired level is achieved.

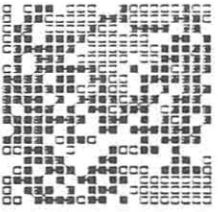
When a cell receives the message from the father cell, the cell tries to increase the level of the left subtree by one. If it is successful, the cell tries to increase the level of the right subtree by one. If it is successful, the level of the tree has been increased by one, and the cell returns the message to its father. But if it fails, the cell is changed into a connection cell, the right subtree is retracted, and the tree expansion is tried at the left subtree again. If the level of the left subtree cannot be increased, the left subtree is retracted, and the cell is changed to the connection cell, and the tree expansion is tried at the right subtree again. The tree is expanded in breadth-first order. Figure 11 shows the trees embedded in a defective array. Since the average degree of the tree graph is three, configuration of cells into the tree in the square, hexagonal, and octal array could be efficient even when working probability of a cell is not high. Table 1 shows the maximum level of the tree into which cells are configured. Note that to increase the level of a tree by one, the number of computation cells in the tree should be multiplied by two. Figure 12 shows the percentage of working cells that are connected into a tree. When the level of the tree does not increase as the working probability increases, the percentage decreases as more working cells are available. Table 2 shows the average delay of the configuration,

As the number of neighbors in the array increases, and as the working probability increases, efficiency of the configuration increases, and average delay decreases as can be expected.

We compared the efficiency of our configuration algorithm with the embedding of H-tree in a defectless square lattice. H-tree is a complete binary tree embedded in a recursive pattern that looks like the letter 'H' [4]. H-tree is known to be the most efficient way of embedding a complete binary tree in a square lattice. The maximum level of H-tree that can be embedded in a defectless square lattice is 9 in a 40 by 40 lattice, and 11



(a) level 6 tree
on octal array
p = 0.7

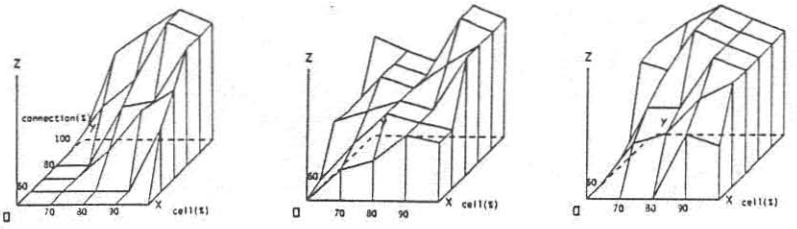


(b) level 7 tree
on octal array
p = 0.7

Figure 11: Configuration of cells into trees in the defective array.

connection		cellular array				
type	probability (%)	cell (%) 60	70	80	90	100
square array (c=4)	50	...	...	...	...	...
	60	...	...	...	...	7.0
	70	...	...	5.0	7.0	7.5
	80	...	...	6.5	7.0	8.0
	90	...	4.5	7.0	8.0	8.0
	100	...	7.0	7.5	8.0	8.0
hexagonal array (c=6)	50	...	...	...	7.0	7.0
	60	...	5.5	6.5	7.5	8.0
	70	...	6.5	7.0	8.0	8.0
	80	...	7.0	8.0	8.0	8.0
	90	6.5	7.0	8.0	8.0	8.0
	100	7.0	7.5	8.0	8.0	8.0
octal array (c=8)	50	...	6.0	7.0	8.0	8.0
	60	3.5	7.0	8.0	8.0	8.0
	70	7.0	7.5	8.0	8.0	9.0
	80	7.0	8.0	8.0	8.5	9.0
	90	7.0	8.0	8.0	9.0	9.0
	100	8.0	8.0	8.0	9.0	9.0

Table 1: Size of the embedded tree.



(a) square array (b) hexagonal array (c) octal array

Figure 12: Percentage of working cells that are connected into a tree.

connection		cellular array				
type	probability (%)	cell (%) 60	70	80	90	100
square array (c=4)	50	...	...	...	...	...
	60	...	...	...	...	2.33
	70	...	...	3.08	2.61	2.47
	80	...	...	2.53	2.51	2.23
	90	...	1.81	2.69	2.35	2.16
	100	...	2.70	2.31	2.25	1.96
hexagonal array (c=6)	50	...	...	...	2.44	2.01
	60	...	2.43	2.70	2.07	2.01
	70	...	2.26	2.04	2.21	1.92
	80	...	2.36	2.04	1.93	1.78
	90	2.52	1.98	1.87	1.78	1.69
	100	2.09	1.81	1.92	1.81	1.61
octal array (c=8)	50	...	2.22	2.07	2.09	1.90
	60	1.80	2.16	2.09	1.84	1.87
	70	2.08	2.06	1.96	1.79	1.90
	80	2.04	2.04	1.83	1.80	1.73
	90	1.98	1.95	1.70	1.84	1.67
	100	1.87	1.80	1.66	1.75	1.58

Table 2: Delay on the embedded tree.

in a 80 by 80 lattice, and the efficiency of embedding the largest H-tree on lattice of size 40 by 40, or 80 by 80 is 32%, and the delay on the H-tree is about 3.4 [13]. Comparing these with table 1 and table 2, we see that we can configure cells into a defective array without much penalty even when there are many defective cells in the array.

4.3 Mesh

On the mesh of cells, every cell has four neighbors: left, right, up, and down. The working cell on the defective array are configured into a mesh by setting its Connection Registers CR[0..3] to the directions of the neighbors connected as up, right, down, and left neighbor of the cell.

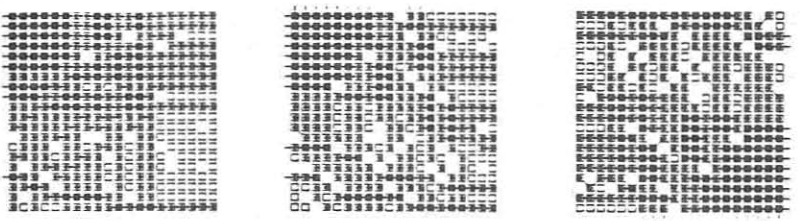
Manning [14] describes the algorithm for embedding a mesh on the defective square array, and Green [8] describes embedding a mesh using the channel between the cells. Both use the knowledge of the global state of the defective array. Here a distributed algorithm where each cell knows only the state of the neighbors (working or defective) is described.

The Controller sends a message to the cell at a boundary of the massively fault-tolerant cellular array, where the message tells the cell to grow a horizontal line of the mesh. If it is successful, the Controller sends a message to the cell at the other boundary to grow a vertical line of the mesh. Growth of the horizontal and vertical line alternates until no more lines of the mesh can be grown on the array. The cells at the junction of a horizontal line and a vertical line become the nodes of the mesh. The cells at the nodes of the mesh are computation cells, and the cells connecting the nodes are connection cells. The computation cells are given the coordinates of the mesh.

Since the complexity of the mesh is four, and the degree of the defective square array is less than four, efficiency of the configuration may be low on the square array. When a growing horizontal line comes across a defective cell, the line should veer around the cell, and this uses the cells which can be used for a vertical line. Veering around the defective cell on the square array while growing a horizontal line blocks the growth of a vertical line, and vice versa. Therefore, bending the line should be done sparingly. On the hexagonal and octal array, the growing horizontal line can use the connection without occupying the cell in the other direction. This increases the efficiency of configuring the cells into a mesh on hexagonal and octal array.

When the cell receives the messages, it tries to grow in the direction of the line. When the neighbor on the direction of the line is defective, it tries to grow on the direction specified on the message. If the cell cannot grow the line, it backtrack to its predecessor cell. Figure 13 shows the cells configured into a mesh.

Figure 14 shows the size of the mesh as the percentage of the array size, and table 3 shows the size of the embedded mesh. and table 4 shows the delay of the configuration. As shown in the figure 13 and table 4,

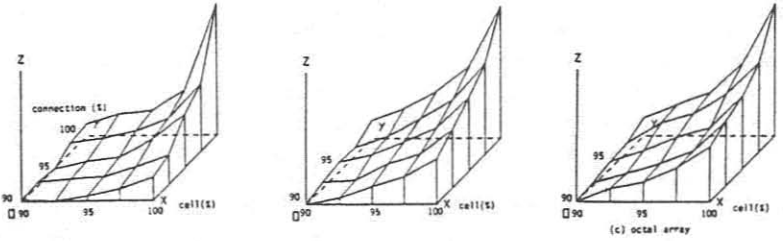


(a) square array
p = 0.95

(b) hexagonal array
p = 0.9

(c) octal array
p = 0.9

Figure 13: Configuration of cells into meshes in the defective array.



(a) square array

(b) hexagonal array

(c) octal array

Figure 14: Percentage of working cells that are connected into a mesh.

the efficiency of the configuration increases rapidly and the delay decreases rapidly with the increase of the number of the neighbors.

5. Conclusion

As shown in this paper, self-configuration of cells into various computation graphs can be done efficiently when cells are adequately powerful. This paper presents the data for various working probabilities of cells and connections with various degrees of the computation graph and of the array. When we use the massively fault-tolerant cellular array for a particular application, we need to configure cells into a particular computation graph, and we can determine the defect rate of cells which allows acceptable efficiency of configuration from the data in this paper. We can change this acceptable defect rate by changing the interconnection patterns or the size of a cell to find the feasible implementation on available technology. Currently we are designing a wafer-scale signal processing chip using the massively fault-tolerant cellular array. The architecture is very homogeneous and simple, and shows the potential for high performance.

connection type	probability (%)	40 × 40 array cell (%) 80	90	95	100	80 × 80 array cell (%) 80	90	95	100
	80	2.0	3.3	5.4	7.5	3.0	6.0	11.5	17.0
square	90	3.3	7.8	8.8	13.3	6.5	11.5	18.0	23.3
array	95	4.3	7.5	10.3	20.8	7.2	13.3	21.3	35.8
(c=4)	100	5.3	10.5	13.8	40.0	8.0	18.5	29.0	80.0
	80	1.8	4.5	7.4	10.3	1.5	9.5	15.0	20.5
hexagonal	90	4.5	10.0	14.0	20.3	7.0	18.8	23.0	38.3
array	95	6.5	12.3	16.8	26.8	10.0	24.0	31.5	48.3
(c=6)	100	6.8	14.3	20.8	40.0	13.0	31.0	43.5	80.0
	80	2.8	9.8	13.4	17.0	6.0	20.0	27.2	34.5
octal	90	5.5	13.0	17.0	24.3	9.5	23.0	31.0	46.0
array	95	6.7	14.5	19.5	28.0	12.7	29.0	39.8	55.0
(c=8)	100	8.0	16.8	22.0	40.0	16.0	32.5	47.3	80.0

Table 3: The size of the embedded meshes.

connection type	probability (%)	40 × 40 array cell (%) 80	90	95	100	80 × 80 array cell (%) 80	90	95	100
	80	21.21	11.11	8.77	6.44	32.17	16.51	11.17	5.83
square	90	13.53	5.94	5.14	3.31	15.74	8.25	5.23	3.94
array	95	11.13	5.98	4.46	2.06	13.64	6.99	4.26	2.43
(c=4)	100	8.73	4.17	3.24	1.00	11.55	4.86	3.03	1.00
	80	15.62	7.53	5.60	3.67	34.71	7.73	5.75	3.77
hexagonal	90	7.81	4.16	2.75	1.93	10.13	4.12	3.38	2.06
array	95	6.56	3.24	2.31	.1.48	7.96	3.25	2.51	1.65
(c=6)	100	5.31	2.73	1.89	1.00	5.79	2.54	1.82	1.00
	80	11.15	3.92	3.10	2.28	11.57	3.86	3.07	2.28
octal	90	6.51	2.99	2.28	1.63	7.73	3.38	2.55	1.73
array	95	5.55	2.67	2.00	1.42	6.24	2.70	1.99	1.45
(c=8)	100	4.59	2.33	1.79	1.00	4.76	2.42	1.68	1.00

Table 4: The delay on the embedded meshes.

References

[1] R. Aubusson and I. Catt, "Wafer-scale integration - a fault tolerant procedure," *IEEE Journal of Solid State Circuits*, **SC-13** (1978) 339-344.

[2] G. H. Barnes, "The Illiac IV computer," *IEEE Transactions on Computers*, **C-17** (1986) 746.

[3] K. E. Batcher, "Architecture of a massively parallel computer," *Proc. 7th Annual Symposium on Computer Architecture*, pp. 168–174, 1980.

[4] R. P. Brent and H. T. Kung, "On the area of binary tree layouts," *Information Processing Letters*, **11** (1980) 46–48.

[5] M. B. Duff, "Review of the CLIP image processing system," *Proceedings National Computer Conference*, pp 1055–1060, 1978.

[6] John W. Essam, "Percolation theory," *Reports on Progress in Physics*, **43** (1980) 833–912.

[7] D. Fussel and P. Varman, "Fault tolerant wafer-scale architectures for VLSI," *Proceedings 9th Annual Symposium on Computer Architectures*, pp. 190–198, 1982.

[8] J. W. Greene and A. El Gamal, "Configuration of VLSI arrays in the presence of defects," *Journal of the ACM*, **31** (1984) 694–717.

[9] J. Hoshen, P. Klymko, and R. Kopelman, "Percolation and cluster distribution. III. Algorithm for site-bond problem," *Journal of Statistical Physics*, **21** (1979) 583–599.

[10] Israel Koren, "A reconfigurable and fault tolerant VLSI multiprocessor array," *Proceedings 8th Annual Symposium on Computer Architectures*, pp. 425–442, 1981.

[11] J. K. Illiffe, *Advanced Computer Design*, (Prentice-Hall, 1982).

[12] T, Kondo, T, Nakashima, M. Aoki, and T, Sudo, "An LSI adaptive array processor," *IEEE Journal on Solid State Circuits*, **SC-18** (1983) 147–156.

[13] Myoung Sung Lee, "Self-configuration of the massively defective cellular array," Ph. D. Dissertation, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, 1986.

[14] F. Manning, "An approach to highly integrated, computer-maintained celluar arrays," *IEEE Transactions on Computers*, **C-26** (1977) 536–552.

[15] B. H. McCormick, "The Illinois pattern recognition computer, ILLIAC III," *IEEE Transactions on Computers*, **EC-12** (1977) 791–813.

[16] C. Mead and L. Conway, *Introduction to VLSI Systems*, (Addison-Wesley, 1980).

[17] Vindo K. S. Shante and Scott Kirkpatrick. "An introduction to percolation theory," *Advances in Physics*, **20** (1971) 325–357.

[18] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The SOLOMON computer," *Proceedings Western Joint Computer Conference*, pp. 87-107, 1962.