ANALYSIS OF ALGORITHMS FOR THE CONFIGURATION OF WAFER SCALE LINEAR ARRAYS IN THE PRESENCE OF DEFECTS

Dimitris A. Doukas
Andrea S. LaPaugh

CS-TR-128-87

December 1987

ANALYSIS OF ALGORITHMS FOR THE CONFIGURATION OF WAFER SCALE LINEAR ARRAYS IN THE PRESENCE OF DEFECTS†

Dimitris A. Doukas
Andrea S. Lapaugh

Computer Science Department
Princeton University
Princeton, NJ 08544

Abstract: Wafer Scale Integration(WSI) is a new technology using Very Large Scale Integration(VLSI). The goal is to implement an entire system in a single silicon wafer containing the equivalent of hundreds of present-day chips. Given the high density and large number of elements in a wafer scale system, we expect some components to be defective due to fabrication errors. Methods are needed to configure the good components into a working system.

In this paper we examine four algorithms to connect the good elements in a linear WSI systolic array. We present experimental results obtained by simulation of the algorithms. We also present an analysis of performance which is applicable to three of the algorithms. The analysis is in term of the number of good elements we expect to utilize.

† This work was supported by DARPA Contract N00014-82-K-0549

I. Introduction

Wafer Scale Integration (WSI) is a new fast developing *VLSI* technology. The goal is to assemble an entire system on a single wafer, thus avoiding the costs and performance loss associated with individual packaging of chips because of long wire connections driven outside the chip.
However new problems arise. Since now we have an entire system in a single wafer, there is a high probability some of the system parts will be defective. A new area of research has developed to deal with these problems.

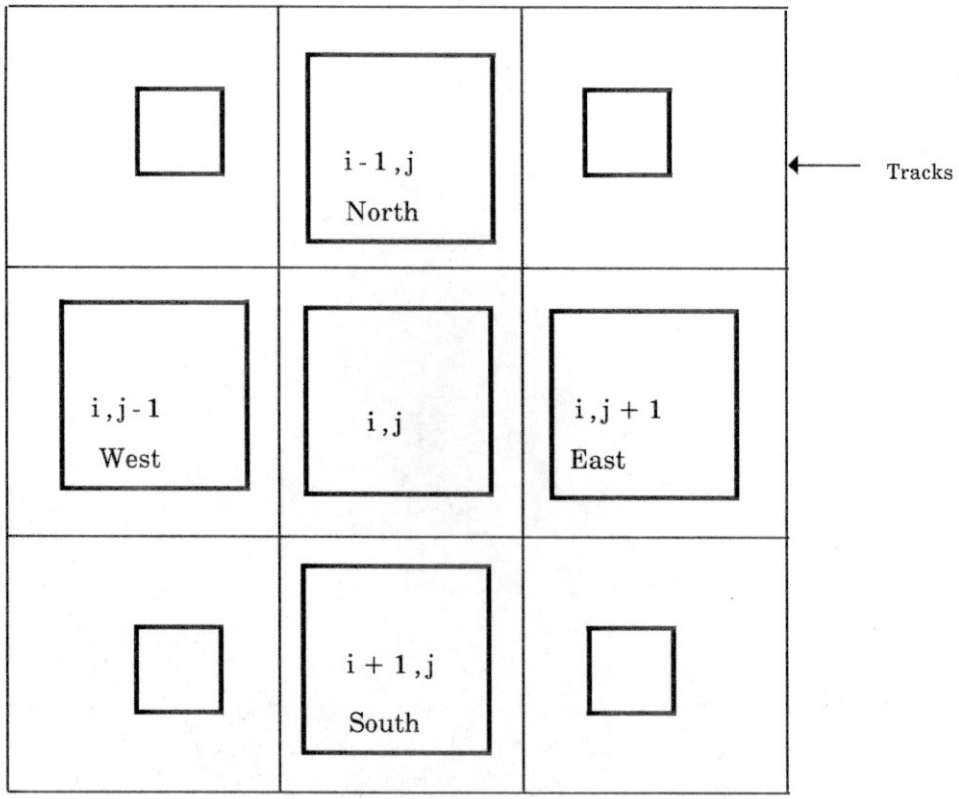
We have not yet specified what we mean by "entire system", what this system includes and how complicated this system may be. Regular architectures are well suited for implementation using *WSI*. The regular structure simplifies implementation. Furthermore they have many applications such as systolic arrays. Systolic arrays are used to solve algorithmically specialized problems such as band matrix multiplication, systems of linear equations, LU decomposition [4]. Therefore we will consider an "entire system" to be a regular array structure; each cell in the array can be considered as a simple processor or a more complicated structure.

In this report we describe four algorithms for the configuration of wafer scale linear arrays in the presence of defects. These algorithms are based on algorithms proposed in the literature.
In *part II* we give a description of each algorithm .
In *part III* we present and analyze the empirical results, comparing the four algorithms.
In *part IV* we estimate the expected utilization for these algorithms using a probabilistic model of cell failure, and compare the expected numbers to what we found experimentally.

II. Description of four algorithms applicable in linear systolic arrays

Before describing each algorithm we give some definitions. A *wafer* is a two-dimensional (2D) array of identical cells. A cell is *dead* if it is defective and *live* otherwise. In Figure II.1 the form of the array is shown. Since the array is structured in a two dimensional grid, we can refer to each cell of the array using a pair (i , j). Each cell has four neighbors, and we name them *n (north), e (east), s (south), w (west)*. Clearly if the cell in the center is the (i , j) cell then the *north, east, south, west* neighbors are respectively the cells $(i -1,j)$, $(i,j+1)$, $(i+1,j)$, $(i,j -1)$. Tracks (wires) are provided between cells to form the necessary frame to make the connections between them. Signals can be routed through these tracks using for example laser restructuring techniques [8]. The number of tracks needed between the cells is dependent on the reconfiguration algorithm we use, as we will see later. We wish to add connections, using the tracks, to form a linear array out of the good cells.

We define the *yield* of the array as the percentage of the fault free cells. By saying that the expected yield is 80% we mean that in a wafer of 100 cells we expect to find 80 live and 20 dead.



Linear array of cells structured in a 2D grid

Figure II.1

To simulate a given yield in our implementation we use a random number generator of the UNIX system [5] in the following way. We assign a number n between *1..100* randomly to each cell. If the desired yield is b% (b integer) we call a cell dead if $n \leq b$ and live if $n > b$ (In section *III* we briefly discuss the quality of this random number generator.)

An important factor in interconnecting a grid of wafers is the maximum wire length of connections because long wires cause delays in a circuit. We chose a maximum allowable wire length. For certain choices of wire length, we do not expect to connect all the live cells in the wafer. To be able to say how many of the live cells we can connect (utilize) we define utilization in the following way:

Utilization = (total # of used cells) / (total # of live cells)

The yield, the utilization, and the maximum length of the wires used to connect the live cells are used to characterize the algorithms.

II 1. The "snake" algorithm

The idea behind the snake algorithm is very simple. It comes from the observation that an obvious way to trace the whole array is to go in a snake - like movement around the rows of the array. When the yield of the array is *100%*, using this method we can go through all the cells, giving *100%* utilization. Of course in the real world we have to face more complicated situations when some of the cells are dead. Then we are allowed to skip over bad cells until we find a good one. However, we must also consider the maximum wire length of connections to be tolerated. We determine a target distance before the execution of the algorithm. Each time we meet a block of dead cells for which the total number of dead cells is bigger than the target (assuming unit distance between cells), we do not allow the connection to be made.

We summarize the algorithm:

We start from the first cell in the first row going east. We connect each cell to the next live cell if the number of the dead cells we skip is not bigger than the target. When we reach the end of the row we begin tracing the next row using the opposite direction. Each row has a predetermined direction, east for the odd numbered rows and west for the even numbered rows. When the connection is not allowed because a bad block of dead cells is bigger than the target, we try to go to the south neighbor. If it is live we continue the same direction. If it is dead, then we backtrack. This is done with a new operation, the *unmark* operation. Since we cannot go anywhere from the cell, we unmark it (in other words we treat it as a dead cell) and we backtrack to the nearest connected neighbor. From this cell we again try to find a path to the last row or the last live cell by first going south, calling the above procedure recursively. If we succeed in building a path to a live cell in the last row or the last live cell in the array, then we say that we successfully connected the live cells in the wafer and we calculate the utilization. If we cannot, then we say that our operation has failed and that the utilization is *0%*. In Figure II.2 we can see how the algorithm works in a *6X6* wafer. Since we build a linear array, each live cell is connected to the next live cell we meet moving in a certain direction (depending also on the target distance). We never initiate a second connection, between another pair of live cells, before the first one is finished. Thus only one track is needed between cells, both vertically and horizontally, to do the reconfiguration using the snake algorithm. For the perimeter of the wafer we need one track in each side except the bottom one (Figure II.2).

We have not discussed how the algorithm works at the boundary of a row. When we first implemented the snake, we did not allow connections over the end of a row, regardless of the length of the series of bad cells we met. This feature is inherently prohibitive to *100%* utilization. So we

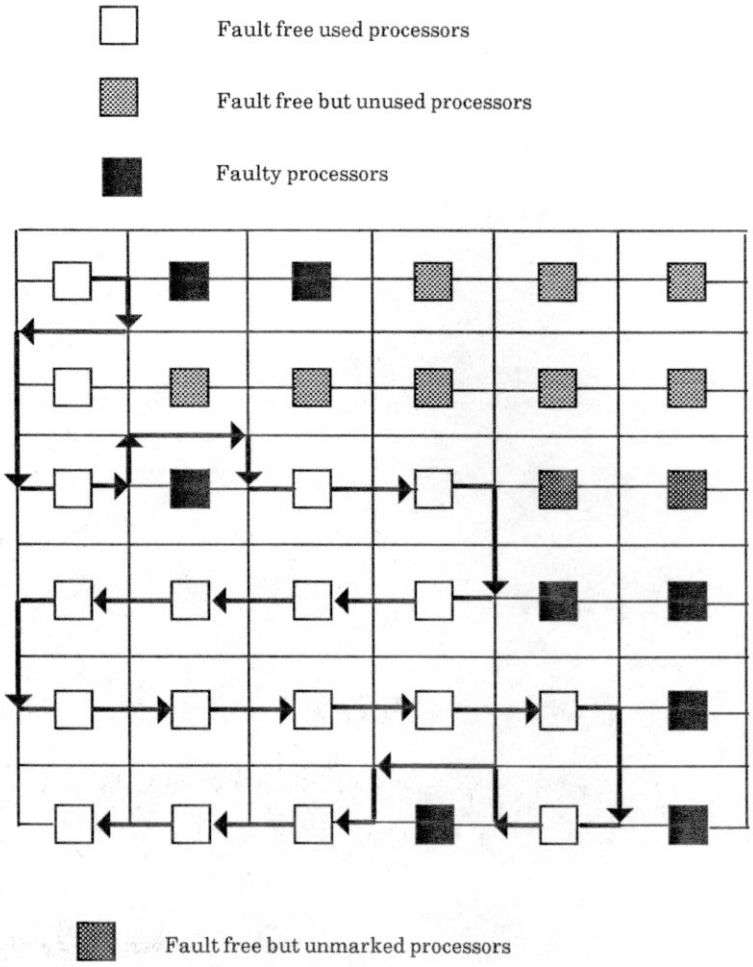


Figure II.2

relaxed the algorithm to allow it to wrap around the ends. In section *III*, we give a graphic plot of the implementation of both versions of the snake. The plot using the first version of the algorithm is for different probabilities of failure of the wafer. Observe that we never succeed in a *100%* utilization even when the yield is *90%*.

The idea behind the snake algorithm was used in [3] by Coren. However he does not allow skipping over bad cells and he provides only one step backtracking. Therefore, for low yield this algorithm gives a very low utilization.

II 2. The "adaptive-snake" algorithm

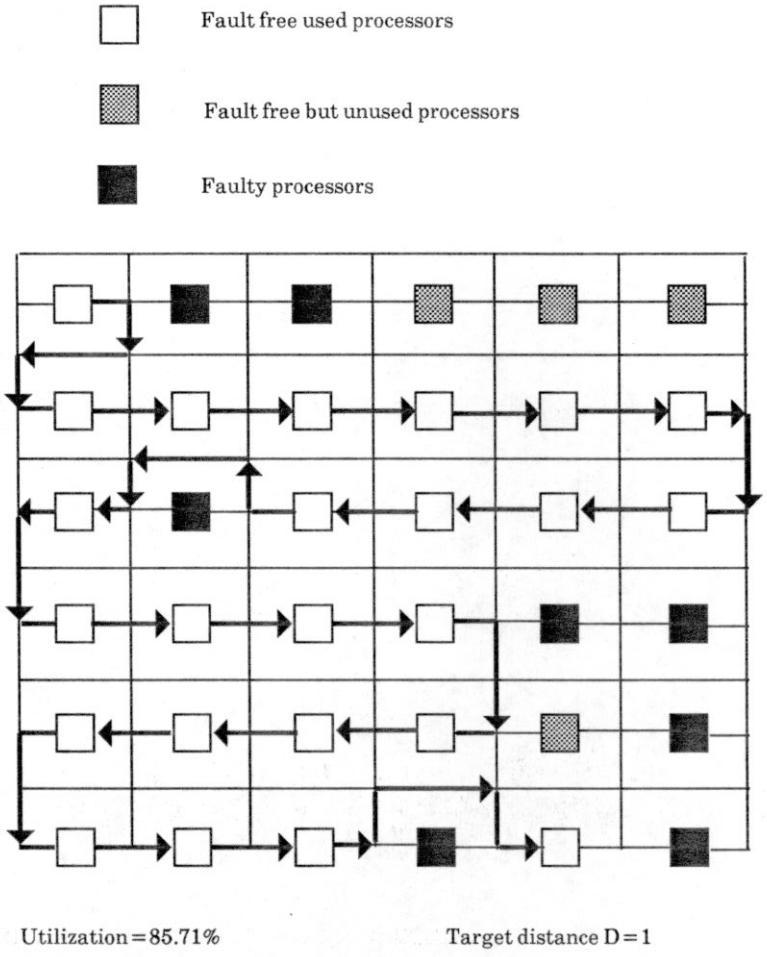


Figure II.3

The adaptive-snake eliminates the restriction that each row has a preassigned direction. It adapts the direction to be chosen depending on the position in the array, and, more specifically, on the i coordinate of the cell. If the wafer is an NxN array, then if $i > N/2$ we go west, otherwise we go east. This means that each time we choose the direction which enables us to go over more cells in a row. The details of the algorithm are exactly the same as those of the snake algorithm.
In Figure II.3 we again use a 6x6 wafer to show how this algorithm differs from the previous one.

II 3. The "partitioning in blocks" algorithm

The idea behind this algorithm is that by using partitioning we can minimize the maximum wire length we need to connect the live cells of the wafer. The length will depend directly on the size of the blocks. In our implementation for an *NxN* wafer, we choose the size of the blocks to be: $\sqrt{N}x\sqrt{N}$.

The algorithm first partitions the entire wafer into $\sqrt{N}x\sqrt{N}$ blocks. After doing the partitioning, we apply the snake algorithm to obtain the configuration inside each block. The final step is to connect the blocks together; we again use the snake algorithm to do this, treating the blocks as cells where a dead block is a block which failed to be configured using the snake algorithm. Since we use the snake algorithm for both inside and outside the blocks, again no more than one track is needed between cells to do the reconfiguration inside each block. To do the connections through the blocks we need also an extra track (vertical and horizontal) between the blocks. So finally we will have one track between elements of the same block, horizontally and vertically, and three vertical or two horizontal tracks between neighbor cells of adjacent blocks (Figure II.4). In Figure II.4 we illustrate the algorithm on our 6x6 wafer. The partitioning algorithm is due to Leighton and Leiserson [1].

II 4. The "spanning tree" algorithm [2]

This algorithm, due to J.Greene and A.Gamal [2], connects a chain of live cells in a two dimensional array. It is a good alternative to be compared with the previous three snake-like algorithms.

We begin again with our two dimensional array of identical cells. We partition the wafer into square blocks of the same size, which we call *sites*. We define a *vacant* site to be a block of cells with a number of live cells less than some critical number. An *occupied* site is one that is not vacant. A connected set of occupied sites, together with all adjacent(vacant) sites define a *cluster*. If a large cluster can be found in this wafer then a large fraction of live cells can be connected together, with moderate interconnecting distances. The analysis in [2] gives a nice algorithm to do the connection of the live cells.

Starting from the *NxN* array, construct the array of N^2/b square blocks of b elements each. We choose b so that each block has a high probability of containing at least four live cells. We consider now a block as a site of the array and if the block has at least four live cells, we consider the site occupied. The idea is that if we choose b so that the above probability is high enough, then the probability to find a cluster of occupied sites (blocks) is nearly equal to one, according to Monte Carlo estimates of percolation probabilities in [6] (for percolation processes see [5]). A tree of maximum degree four that spans the cluster, with all nonleaf sites occupied, can be constructed (A spanning tree of blocks). We can form a chain from all the live cells in the cluster by looping around the tree as shown on Figure II.5a (taken from Fig. 11 in [2]). In Figure II.5b we show the structure of the

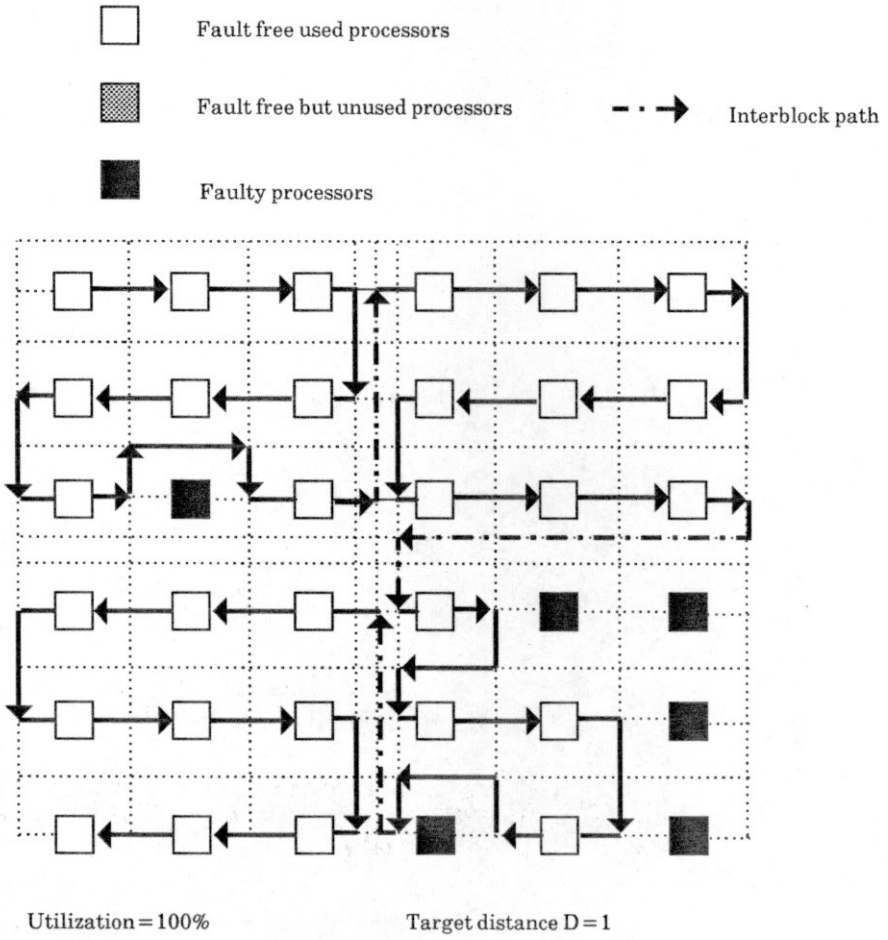


Figure II.4

spanning tree constructed for the wafer of Figure II.5a after using the spanning tree algorithm. The critical point is that since all "nonleaf" blocks have at least four live cells, it is never necessary to connect two cells from nonadjacent blocks, thus keeping the interconnecting distances small. In Figure II.5a we illustrate the algorithm on a *9x9* wafer partitioned in *3x3* blocks. As we see only two tracks are needed between adjacent blocks since only two connections are made between the blocks. Inside the blocks the connection of all live cells requires only one track between cells.

The non-uniform spacing, in terms of the number of tracks between the cells, needed for the spanning tree algorithm was also needed by the partitioning algorithm of section II.3. In order to restructure a wafer using one of the partitioning algorithms, we have to predetermine the size of the

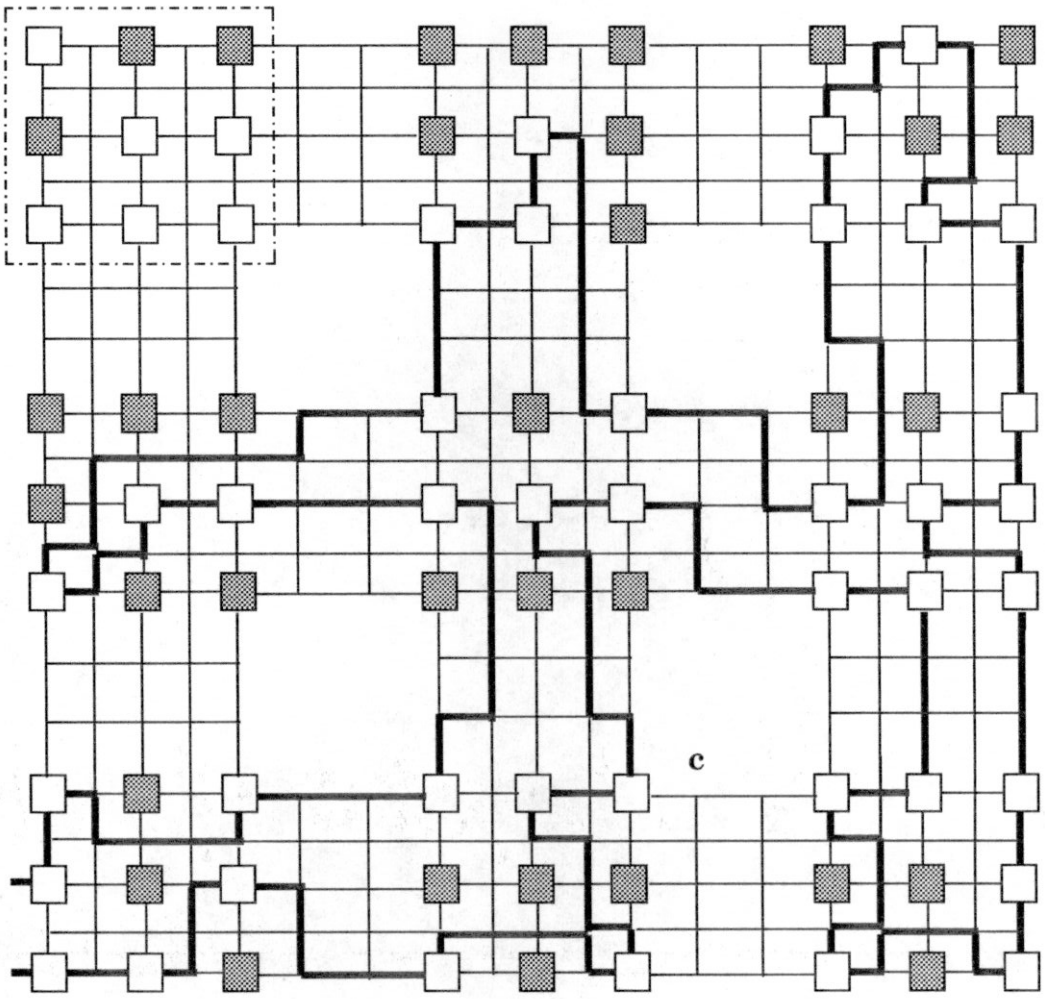


Figure II.5a

(Figure 11 in [2])

blocks in order to build the necessary number of tracks between the cells. In contrast, using the simple snake algorithm simplifies the implementation, since uniform spacing can be used.

Another interesting point of comparison between the algorithms is how the length is controlled. As we saw, for the snake-like algorithms we are interested in the number of consecutive bad cells (bad block) we must bypass in order to make the connections between the live cells (we defined this number as the target distance). In the spanning tree algorithm the main factor which determines the

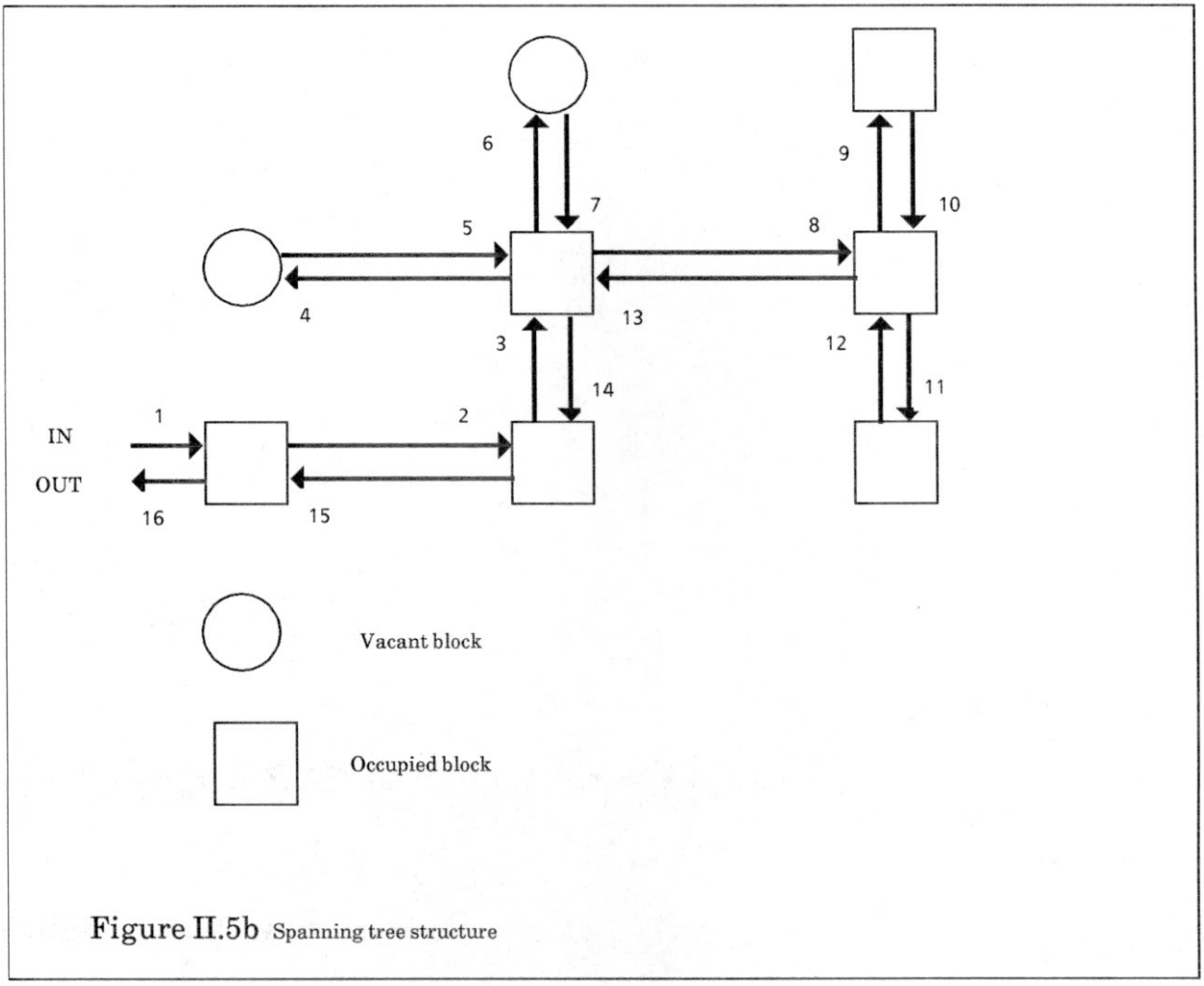
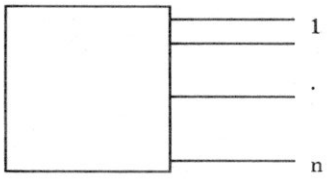
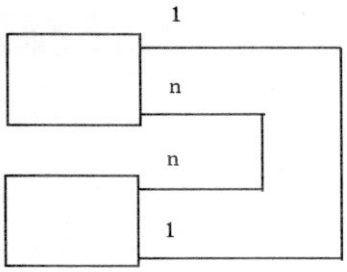


Figure II.5b Spanning tree structure

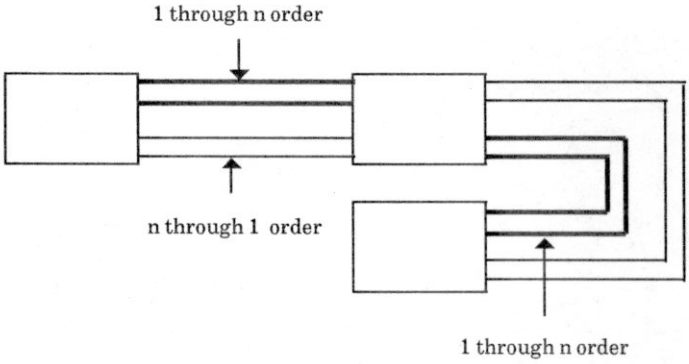
worst case length is the block size. We pointed out previously that it is never necessary to connect two cells from nonadjacent blocks. In Figure II.5a we see that the worst case, in terms of wire length, is when we need to connect cells *a,c*. The wire length needed for this case is $6 \cdot \sqrt{b}-3$ (Manhattan distance, where we assume square cells of size *1x1*), which is dependent on the block size. In the next section, in order to compare the algorithms, we relate the above Manhattan distance with the "target distance" as it is defined for the snake-like algorithms. There is no restriction in terms of length for connections inside the blocks for the spanning tree algorithm. In [2] Green and Gamal do not explicitly state a systematic way to connect the cells inside the blocks. We only have to consider that, for each non-leaf block, two ports (one input and one output) have to be provided in each side. Since we have at least four active elements inside each occupied block this is always feasible. We can assume a snake-like connection between the ports and the rest of the cells inside the blocks.



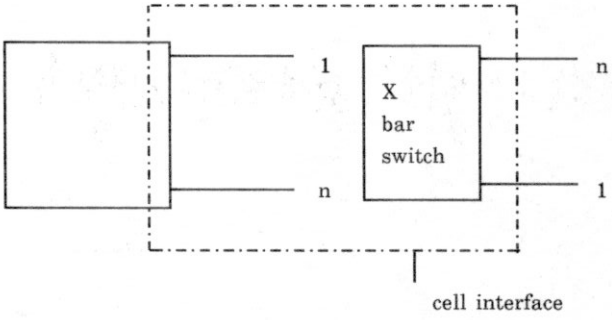
II.6a



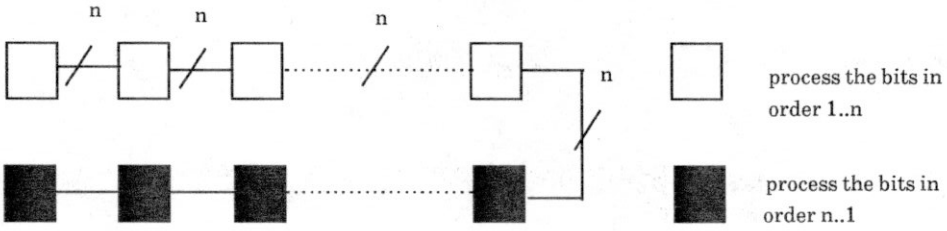
II.6b



II.6c



II.6d



II.6e

Figure II.6

II 5. Routing Details

Let now take a closer look at the tracks. We use these to do the routing between the live cells. When the wafer is operating, information (data) flows through the tracks and it is processed sequentially by each working cell of the linear array. In other words tracks serve as buses where information is being transmitted. Since information is represented in terms of bits, we may wish to transmit in parallel a certain number of bits through the tracks. Therefore a track is a collection of single lines, each of which transmits one bit (Figure II.6a). The *width* of the track is determined by the number of bits transmited in parallel each time. We see in Figure II.6a that there is a virtual ordering of the single lines of the track. This order represents the order in which bits are entered the cells for processing. Since we assumed identical cells this ordering should be preserved as we process the data stream through the linear array. Consider now the case when we snake between two rows of the wafer, as in Figure II.6b. Clearly now the ordering of the bits in the second row is reversed and since we assumed identical cells, the cells of the second row will incorrectly process the entering bits as if they had the initial ordering. Notice that this problem is not particular to the structuring algorithm we use, since in any of these we adopt a snake-like movement. However, the solution to this problem may be dependent on the structuring algorithm.

One solution applicable to all the algorithms is the one we illustrate in Figure II.6c. Here we double the track width by sending the information in both the original and the reverse order. The problem with this solution is that we double the complexity of the wiring.

Another solution also applicable to all algorithms is to use crossbar controllable switches at the cell interfaces, to change accordingly the order of the bits (Figure II.6d). Since a crossbar for n lines has a $O(n^2)$ complexity, this solution is also expensive. In addition it requires control bits for each switch.

The third solution it is well suited to the snake algorithm. Here we take advantage of the fact that in the simple snake algorithm we know the direction we are going to move in each row since we adopt a pure snake movement. In other words, we know that the bit ordering in each odd numbered row will be the original way, and in each even numbered row will be the reverse way. Instead of using one kind of cells we use two, say black and white (Figure II.6e), where all odd rows have white cells and all even have blacks. If we process a sequence of n bits in a white cell and then we process the same sequence in the reverse order in a black cell, the output of the two cells will be the same. Clearly this solution cannot be applied to the adaptive snake algorithm or the spanning tree algorithm since in these algorithms each row does not have a fixed direction. To use this solution with the partitioning algorithm, we must know the block size in advance, so we can assign directions to cell rows. However, since we also need to predetermine the block size in order to build the tracks, this is not an additional loss of flexibility. The last solution does not add complexity in terms of wiring but does

in terms of cell types. Usually for wafer scale implementations the cells are very simple and not so different processing elements, so we can pay the extra cost of fabricating two kinds of them.

III. Analysis of the empirical results

Our experimental work consists actually of two parts. First we ran samples to compare the snake-like algorithms (snake,adaptive-snake,partitioning) and then we ran samples to compare these algorithms with the spanning tree algorithm.

For the first part we experimented with wafers of sizes 16x16, 25x25, 81x81, 100x100, 128x128, 256x256, 512x512. We ran 20 samples of each case and we took the average utilization for yield values 90%, 80%, 70%, 60%, 50%, 40%, 30%. Actually we used as a variable the probability $p\%$ to meet a bad cell which for a wafer of yield $y\%$ is $p = 100\text{-}y\%$. We started with distance 1 as the target length and incremented by one until we found the distance where a 100% utilization was achieved. After these studies, we focused on 50% yield. We experimented with NxN wafers where N is a power of 2, using wafers of size 16x16, 32x32, 64x64, 128x128, 256x256, 512x512.

Figure III.1, shows how the utilization is dependent on the distance for different wafer sizes, when the snake algorithm is applied to them. We see that in all cases we achieve 100% utilization for a certain distance value and this value increases as the size of the wafer is increased. This is because, the number of the bad blocks we expect to meet in a wafer is proportional to the size of the wafer. Another interesting point is that for a NxN wafer 100% utilization is achieved at distance $2 \cdot lgN$, something we expect theoretically (part IV). So for the 128x128 wafer for all 20 samples, 100% utilization is achieved after distances 14 or 13, for the 256x256 the results are 15,16,17 and for the 512x512 between 17 and 19.

Figure III.2, shows the results of the first version of the snake applied on a 81x81 wafer for different yield values. Even for very high yield (10%) we do not achieve 100% utilization. The reason is that in this version of the snake we did not allow wrapping around the corners.

Figures III.3,III.4 compare the three snake-like algorithms for certain wafer sizes by pairs, and for 50% yield. In Figure III.3 the snake and the adaptive-snake are compared and a table gives us the exact values of the utilization for each target distance. As we see the gain after using the adaptive-snake is very small. Since we desire simplicity in our implementations, the snake algorithm seems preferable.

The situation is not the same in Figure III.4, where we compare the snake and the partitioning in blocks algorithms. Here the partitioning algorithm is visibly superior to the snake. For this plot we used samples from 20 121x121 sized wafers and when we applied the partitioning algorithm we partition the wafer into 11x11 square blocks. Since in each block we apply the snake algorithm, this plot is nothing more than a comparison of the snake algorithm applied to two different wafers of sizes

121x121 and 11x11 for 50% yield. Notice the analogy of the results between this comparison and the one we made in Figure III.1. Since, as we will discuss later (section IV), the partitioning algorithm achieves the theoretical lower bound of the interconnection lengths, it turns out to be our first choice as a candidate, especially when the size of the wafer is big enough and the interconnection distance length becomes a critical factor.

For the second part, in order to implement the spanning tree algorithm and compare it with the snake-like algorithms we made our choice of the wafer size depending upon the block size we were going to use. We ran experiments for blocks of size 2x2, 3x3, 4x4, 5x5. In all these cases our requirement was to have in each block at least four live cells. For blocks of size 3x3 we ran experiments on 20 samples of wafers of sizes 12x12, 15x15, 33x33, 63x63, 126x126, 255x255 and we took the mean value of the utilization. For blocks of size 2x2 and 4x4 we experimented on wafers of sizes 12x12, 16x16, 32x32, 64x64, 128x128, 256x256. For blocks of size 5x5 we ran experiments on wafers of sizes 10x10, 15x15, 35x35, 65x65, 130x130, 255x255. In all cases we concentrated on 50% yield.

A critical point for our comparison is how the interconnection distance between two live cells is defined in [2]. Here the distance is defined as the Manhattan distance between the two cells, taken into consideration that one track is required between cells in the same block and two tracks are required between blocks to do the reconfiguration. If we partition the wafer in blocks of size $\sqrt{b}x\sqrt{b}$ the maximum Manhattan distance d required in order to form the chain is $d=6\cdot\sqrt{b}-3$ (We assume square cells of size *1x1*). On the other hand when we talked about target distances in the snake-like algorithms, we meant how many bad cells we have to bypass in order to connect the live cells to form a linear array. In order to compare these algorithms in terms of the utilization and the interconnection distance we have to somehow find a relation between them. This can be done if we look at the worst case. The maximum Manhattan distance given above is required when we have to connect, say, cells *a,c* in Figure II.5a. Speaking in terms of how many bad cells we need to bypass this gives us that we need to bypass $3\cdot\sqrt{b}-3$ dead cells. Finally we have to take into consideration the fact that in order to implement the spanning tree algorithm we need two tracks between the blocks, instead of the three vertical and two horizontal tracks we need when implementing the partitioning in blocks algorithm, and the one track (horizontal and vertical) needed between cells in the implementation of the simple snake algorithm. But assuming that cells are considerably bigger than tracks, we can omit the length penalty of the extra tracks and take the distance $3\cdot\sqrt{b}-3$ as a fair comparison measure between these algorithms.

Figure III.5, just gives the various values of the utilization after applying the spanning tree algorithm to wafers of different sizes in terms of the target distance (not the Manhattan distance). We see that started with distance 4 (2x2) blocks the utilization is 0%. The reason is because we ask, in a wafer with 50% yield, to have 2x2 blocks with four live cells, which is very unlike.

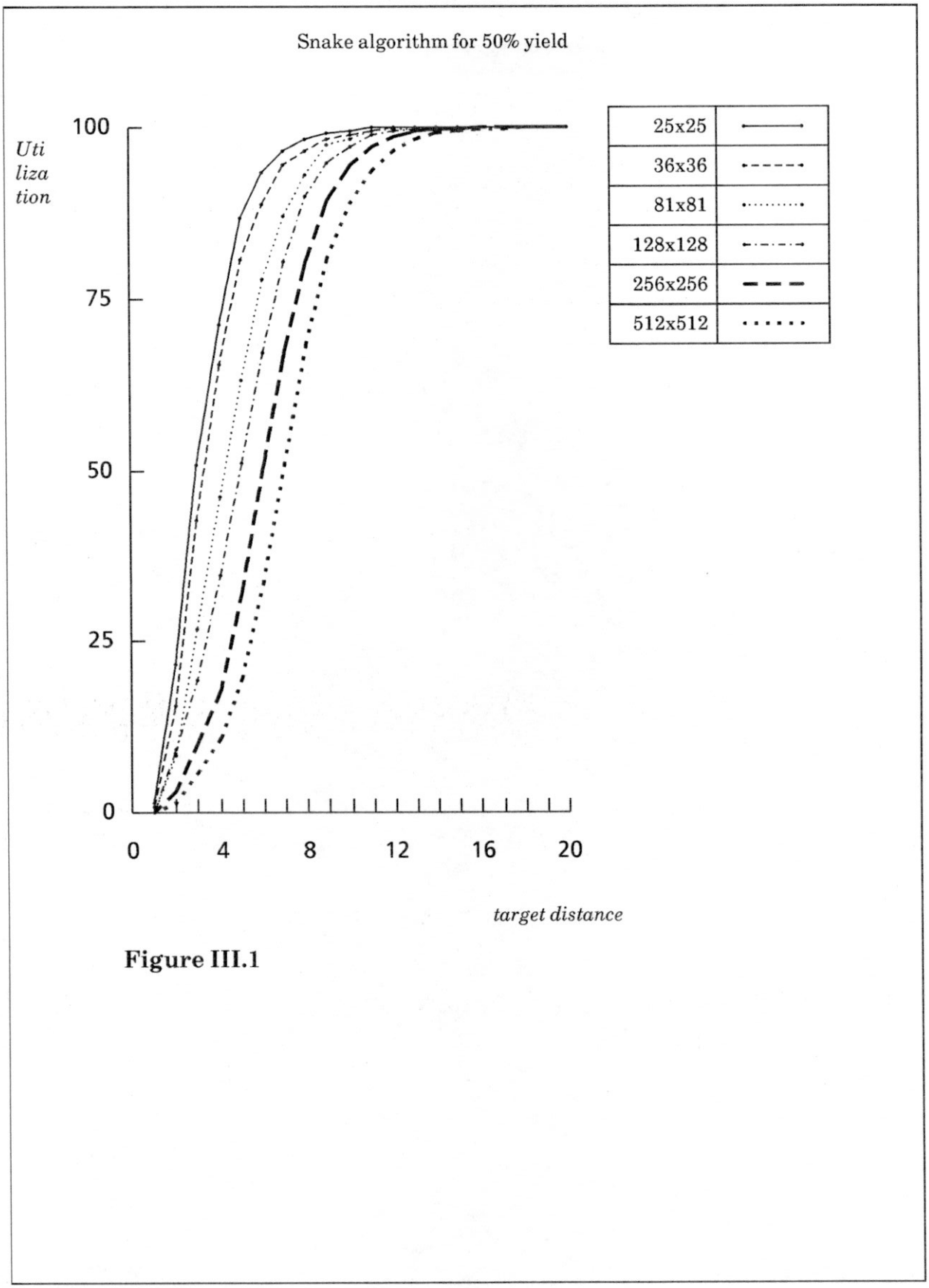

Snake algorithm for 50% yield
Uti
liza
tion
100
75
50
25
0
0
4
8
12
16
20
target distance
25x25
36x36
81x81
128x128
256x256
512x512

Figure III.1

First version of the snake algorithm for a 81x81 wafer

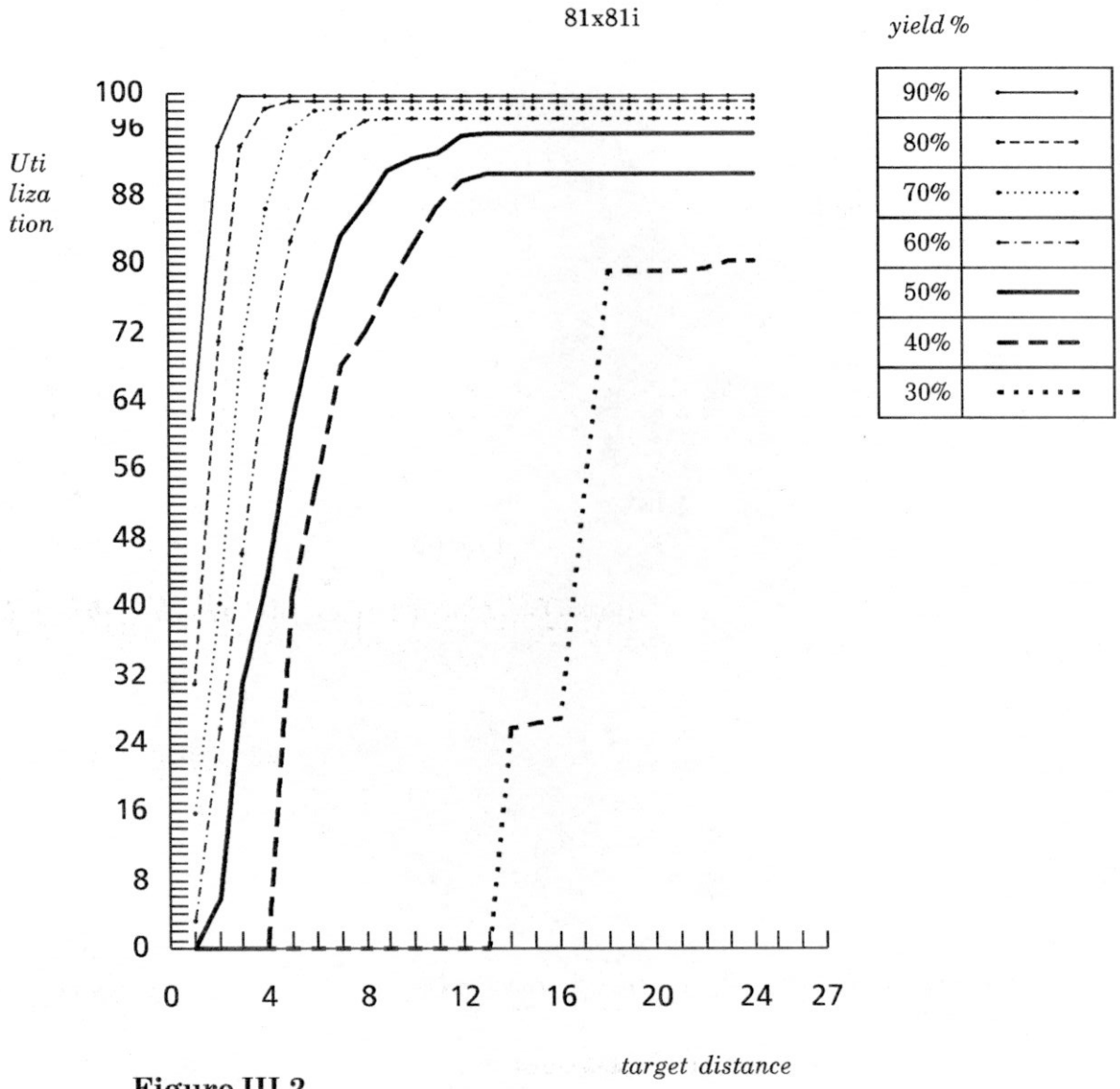


Figure III.2

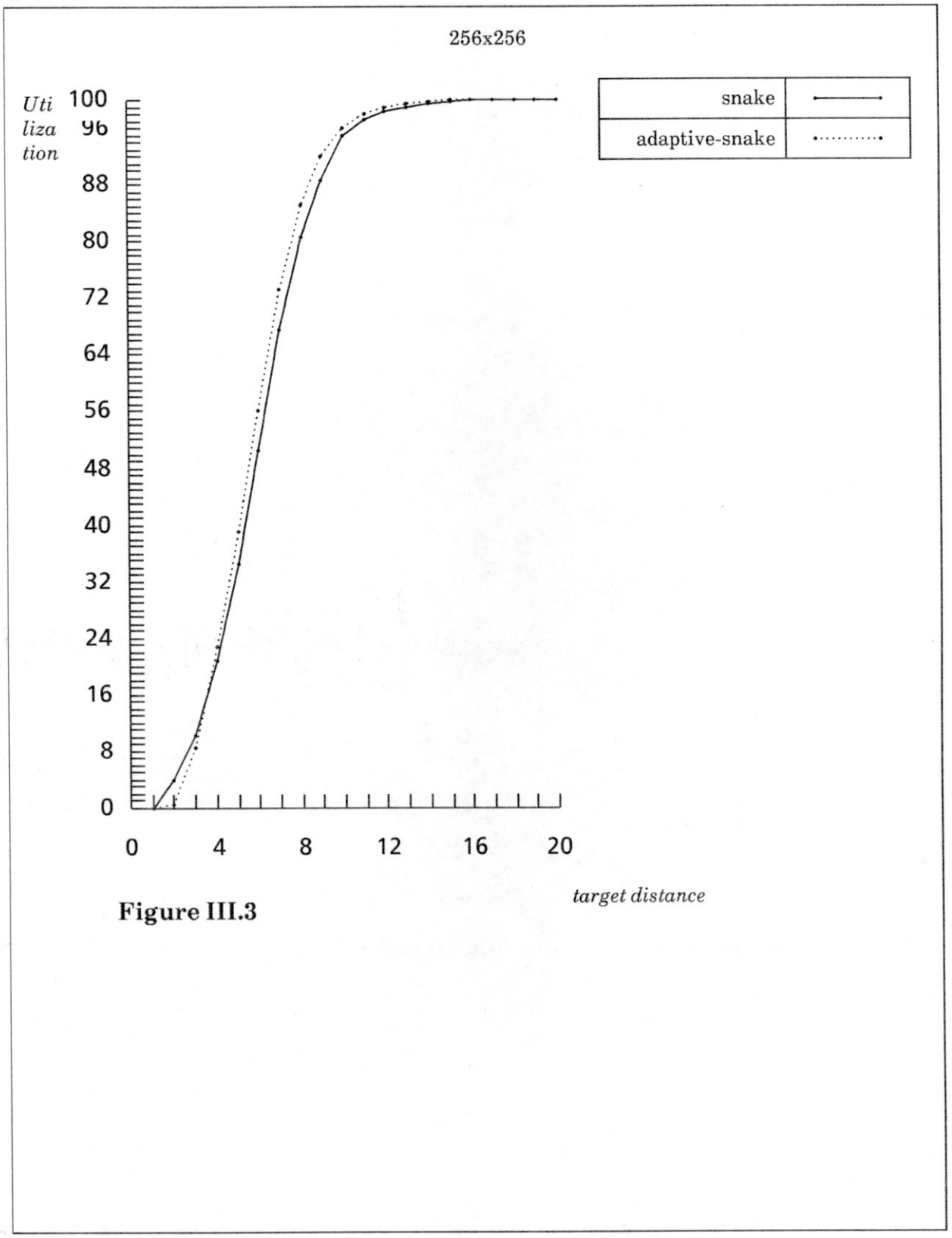

256x256
Uti
liza
tion
100
96
88
80
72
64
56
48
40
32
24
16
8
0
0
4
8
12
16
20
target distance
snake
adaptive-snake

Figure III.3

Target distance for a 256x256 wafer	Snake		adaptive-snake	
	Utilization	Deviation	Utilization	Deviation
1	0.00	0.00	0.00	0.00
2	4.02	2.65	0.51	1.52
3	10.28	3.45	8.59	4.33
4	20.89	1.12	22.87	1.56
5	34.38	1.40	38.99	1.87
6	50.31	2.35	56.17	1.77
7	67.49	2.38	73.35	1.67
8	80.75	1.78	85.13	1.13
9	88.64	2.08	91.96	1.23
10	94.74	1.59	96.12	1.18
11	97.03	1.21	97.97	0.83
12	98.35	1.38	98.98	0.88
13	98.97	1.08	99.41	0.59
14	99.52	0.65	99.69	0.42
15	99.79	0.41	99.86	0.29
16	99.90	0.22	99.93	0.15
17	100.0	0.00	100.0	0.00
18	100.0	0.00	100.0	0.00
19	100.0	0.00	100.0	0.00
20	100.0	0.00	100.0	0.00

Table III.1

The curves rise very sharply at the next distance where 3x3 blocks are defined and now it is very likely to find blocks with at least four live cells. Going one step further to 4x4,5x5 blocks we achieve 100% utilization for all the sizes. Of course as we increment the size of the block the interconnecting distances inside the block are increased but not more than the target distance.

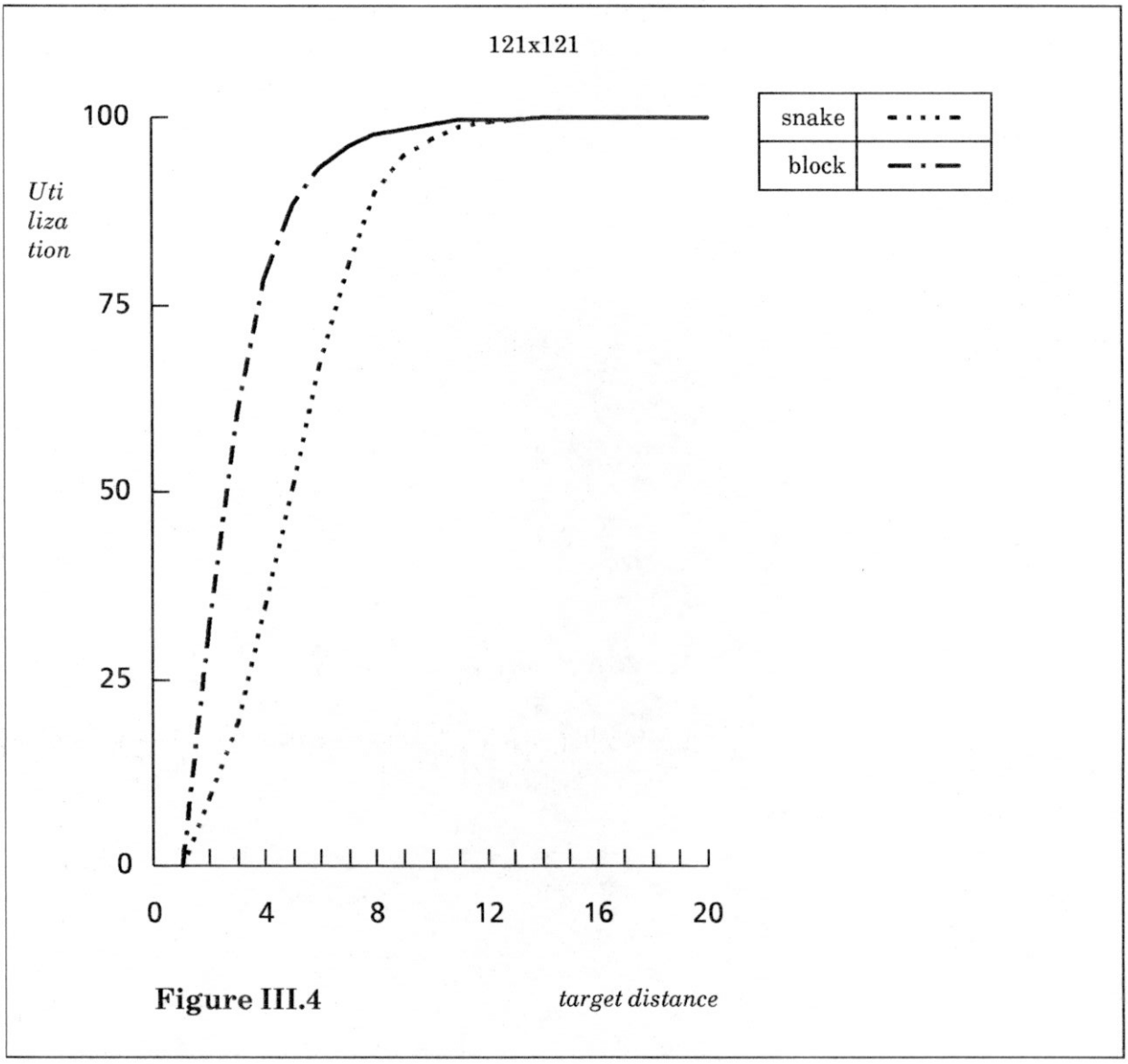


Figure III.4

Tables III.3 and III.4 give analytical results for the snake,partitioning and spanning tree algorithms for different wafer sizes and target distances 3,6,9,12. Finally Figures III.6,III.7 give us some interesting results. In Figure III.6 we compare the snake and the partitioning with the spanning tree algorithm for wafers of size 126x126 (for distances 3,6), 128x128 (for distance 9) and 130x130 (for distance 12). For distances less than 4 where the spanning tree algorithm gives 0% utilization the snake-like algorithms give a low but not zero utilization. After that distance the spanning tree algorithm, does better and achieves faster 100% utilization. The comparison is made clearer if we look also Figure III.7. Here we keep constant the target distance (here 6) and we

Target distance for a 121x121 wafer	Snake		partitioning	
	Utilization	Deviation	Utilization	Deviation
1	0.00	0.00	0.00	0.00
2	9.19	4.62	31.92	4.84
3	19.20	6.68	60.55	2.21
4	34.57	2.31	78.44	2.01
5	50.98	2.55	88.32	1.45
6	67.20	4.81	93.22	1.44
7	80.36	4.61	95.98	1.27
8	90.01	2.96	97.64	1.14
9	94.80	2.45	98.50	0.93
10	97.25	2.02	99.14	0.91
11	98.74	1.28	99.60	0.49
12	99.31	0.78	99.76	0.34
13	99.80	0.36	99.83	0.33
14	99.89	0.09	99.93	0.22
15	99.89	0.09	99.93	0.22
16	99.89	0.09	99.93	0.22
17	100.0	0.00	100.0	0.00
18	100.0	0.00	100.0	0.00
19	100.0	0.00	100.0	0.00
20	100.0	0.00	100.0	0.00

Table III.2

change the wafer size. As we expect the curve for the snake-like algorithms is falling because as we increase the size of the wafer we meet more and bigger bad blocks, and since we keep the target

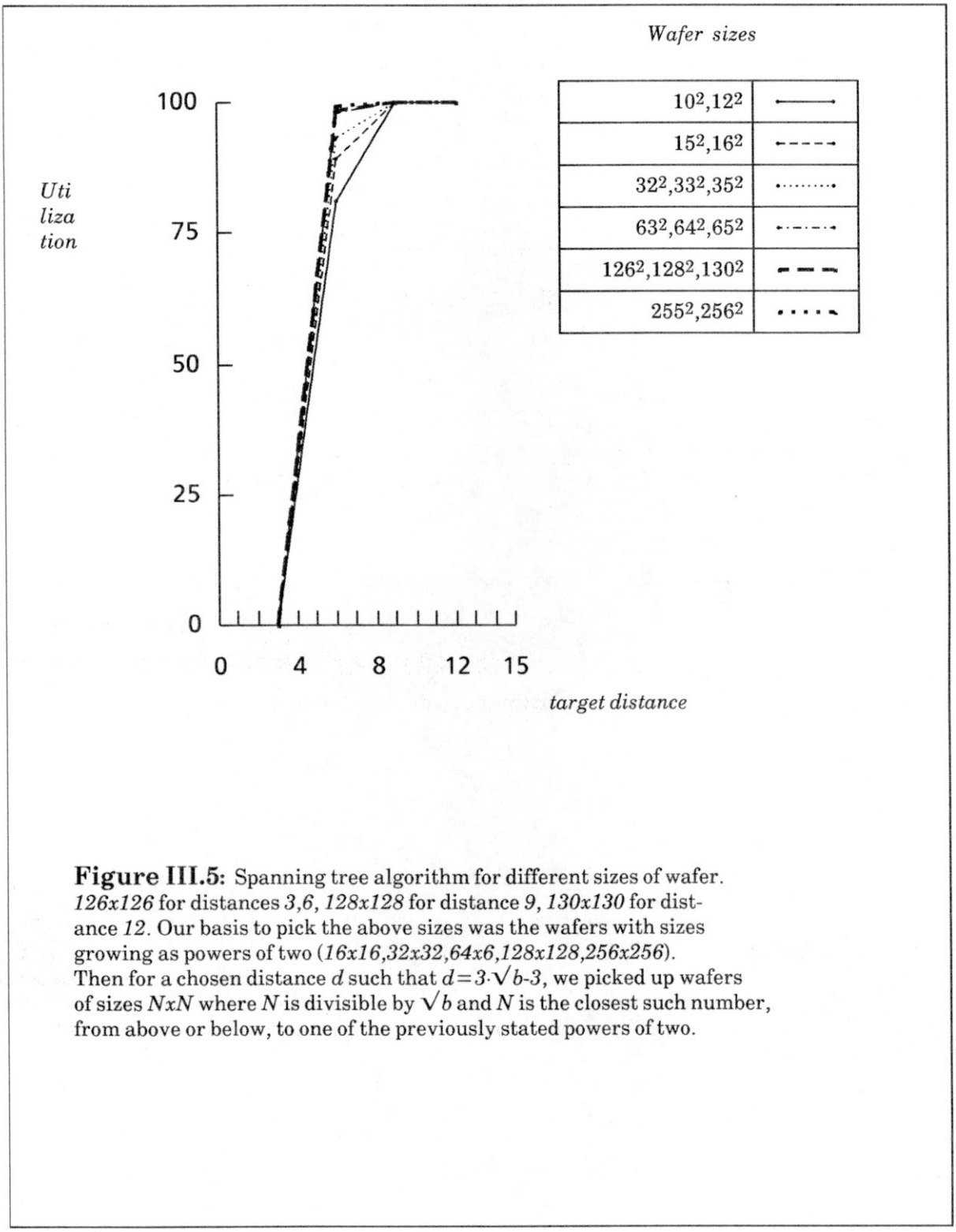


Figure III.5: Spanning tree algorithm for different sizes of wafer. *126x126* for distances *3,6*, *128x128* for distance *9*, *130x130* for distance *12*. Our basis to pick the above sizes was the wafers with sizes growing as powers of two (*16x16,32x32,64x6,128x128,256x256*). Then for a chosen distance d such that $d=3\cdot\sqrt{b}-3$, we picked up wafers of sizes NxN where N is divisible by $\sqrt{b}$ and N is the closest such number, from above or below, to one of the previously stated powers of two.

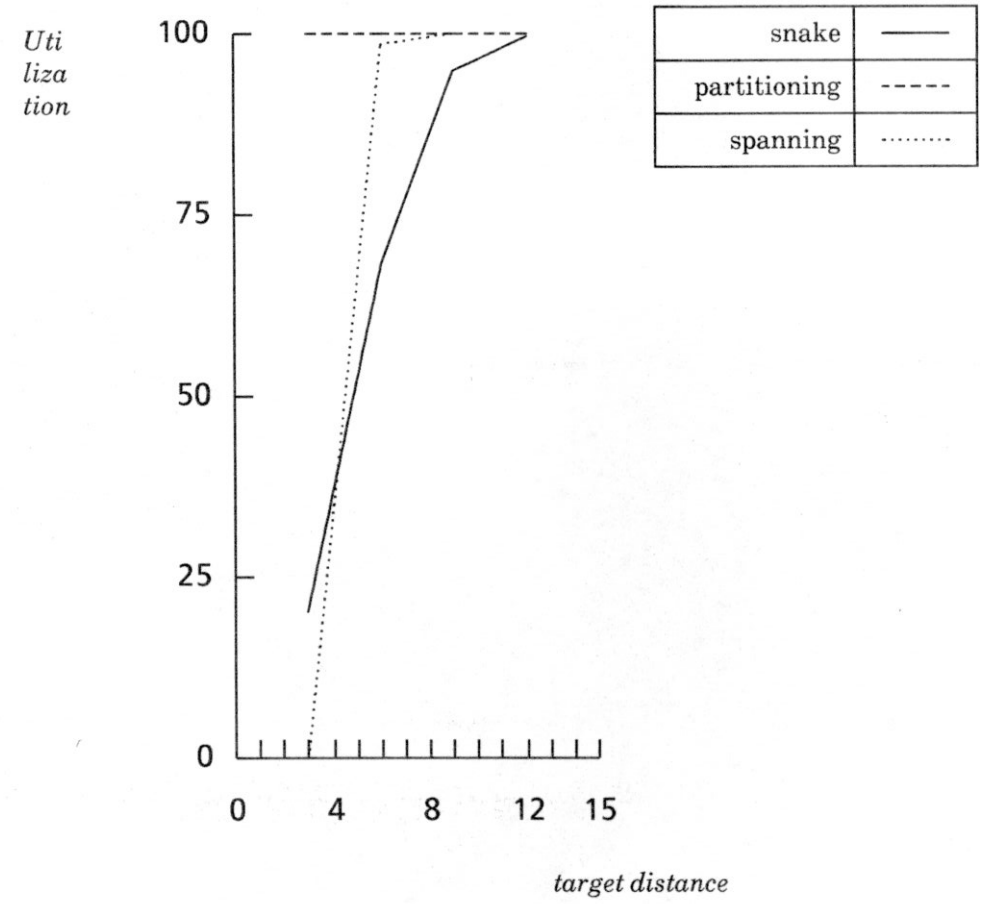


Figure III.6 : Comparison of the three algorithms for different target distances and for wafers of size: *126x126* (distances *3,6*), *128x128* (distance *9*), *130x130* (distance *12*).

distance constant the utilization is decreasing. The opposite thing happens for the spanning tree algorithm. Theoretically for a wafer of infinite size and depending on the value of b, a cluster containing all but a finite number of the good elements, can be found [5] with high probability. So as

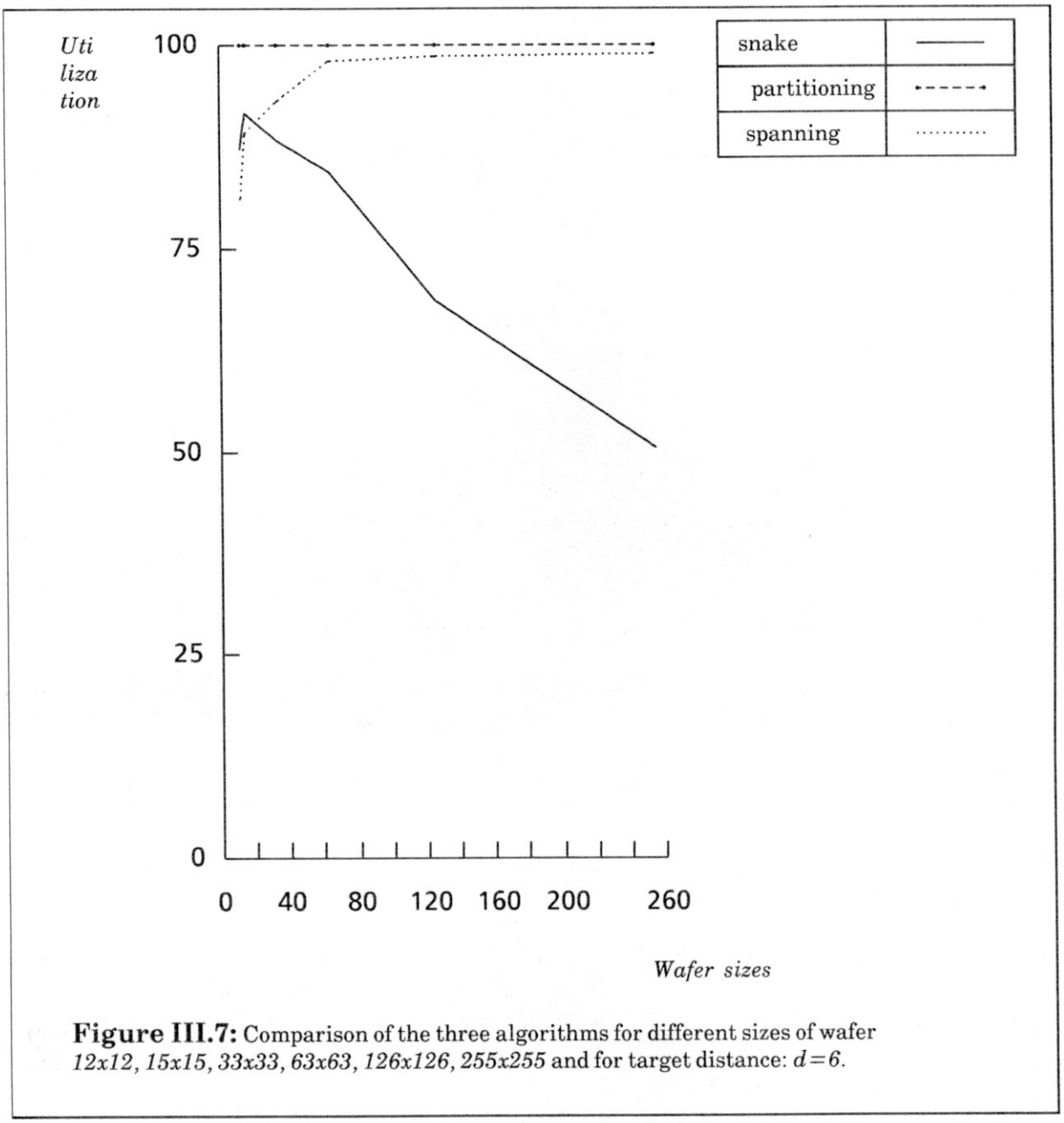


Figure III.7: Comparison of the three algorithms for different sizes of wafer *12x12, 15x15, 33x33, 63x63, 126x126, 255x255* and for target distance: $d=6$.

we increase the size of the wafer in our simulation, we expect to find, for the same distance d, a bigger cluster and therefore a higher utilization.

Partitioning the wafer into the same size blocks, as we did for the spanning tree algorithm, and applying the partitioning algorithm, we obtain 100% utilization for all the different wafer sizes if we use the same maximum allowable distance. Since this distance is $3\cdot\sqrt{b}-3$, it dominates every

Wafer sizes *distance d=6*	Snake		Partitioning		Span. Tree	
	Utilization	Deviation	Utilization	Deviation	Utilization	Deviation
12x12	87.25	29.41	100.0	0.00	81.10	22.86
15x15	91.57	3.33	100.0	0.00	89.30	15.10
33x33	88.18	3.97	100.0	0.00	93.42	8.2
63x63	84.47	3.49	100.0	0.00	98.11	2.1
126x126	68.52	3.70	100.0	0.00	98.50	1.83
255x255	50.40	2.70	100.0	0.00	99.20	0.52

Wafer sizes *distance d=3*	Snake		Partitioning		Span. Tree	
	Utilization	Deviation	Utilization	Deviation	Utilization	Deviation
12x12	70.09	24.87	100.0	0.00	0.00	0.00
15x15	26.90	34.94	100.0	0.00	0.00	0.00
33x33	47.48	16.39	100.0	0.00	0.00	0.00
63x63	27.03	13.95	100.0	0.00	0.00	0.00
126x126	20.16	0.96	100.0	0.00	0.00	0.00
255x255	10.88	0.76	100.0	0.00	0.00	0.00

Table III.3: Comparison of the three algorithms for target distances $d=6,3$

interconnection distance inside the $\sqrt{b}x\sqrt{b}$ size blocks. So in both algorithms which partition the array, we expect to utilize every live cell inside the blocks. The small difference in the utilization we get is because of the different way we connect the blocks. Using the partitioning algorithm we snake around the good blocks to do the configuration. Using the spanning tree algorithm we apply the spanning tree method. In the first case we characterize a block as good if even has only one good cell

Wafer sizes *distance d=12*	Snake		Partitioning		Span. Tree	
	Utilization	Deviation	Utilization	Deviation	Utilization	Deviation
12x12	100.0	0.00	100.0	0.00	100.0	0.00
16x16	100.0	0.00	100.0	0.00	100.0	0.00
32x32	100.0	0.00	100.0	0.00	100.0	0.00
64x64	100.0	0.00	100.0	0.00	100.0	0.00
128x128	99.35	0.58	100.0	0.00	100.0	0.00
256x256	98.64	0.39	100.0	0.00	100.0	0.00

Wafer sizes *distance d=9*	Snake		Partitioning		Span. Tree	
	Utilization	Deviation	Utilization	Deviation	Utilization	Deviation
12x12	88.87	29.81	100.0	0.00	100.0	0.00
16x16	99.69	0.93	100.0	0.00	100.0	0.00
32x32	98.98	1.70	100.0	0.00	100.0	0.00
64x64	98.70	1.92	100.0	0.00	100.0	0.00
128x128	94.33	2.12	100.0	0.00	100.0	0.00
256x256	89.24	1.79	100.0	0.00	100.0	0.00

Table III.4: Comparison of the three algorithms for target distances $d=12,9$

inside. In the second case we require at least four active elements. For small sizes blocks (2x2,3x3) the second algorithm gives a lot (2x2) or some (3x3) dead(vacant) blocks, and this is the reason why we do not obtain a 100% utilization.

Until now, all our comparisons in terms of interconnection length refer to worst case connections. Figures III.8,III.9 give us an idea of how the algorithms compare, in terms of average connection length. We simulated the snake,partitioning and the spanning tree algorithms for different worst

case distances, namely 3,6,9,12,15,18,21,24,27,30 (Figure III.8). We ran samples on wafer of sizes 256x256(distances 3,9,21), 255x255(distances 6,12), 252x252(distances 15,18,24), 250x250(distance 27), and 253x253(distance 30). In Figure III.9 we present runs for the same algorithms but for bigger target distances. We choose the worst case target distance d, to correspond to block sizes (for the partitioning algorithms) growing as powers of two, namely 2x2, 4x4, 8x8, 16x16, 32x32, 64x64, 128x128, 256x256 and we experimented on 256x256 wafers. For the snake algorithm we achieve the best average length. The reason why the snake algorithm is better than the partitioning algorithms is that when we use a partitioning algorithm, the interconnections between the blocks contribute a significant fraction on the average length, proportional to the block size. The curve for the snake algorithm is rising as d increases until we reach average length one and then it becomes constant independently of the d value. The reason is that as we increase the target distance, we are allowed to bypass longer bad blocks, increasing the average connection length. When the target distance d becomes bigger than the longest bad block then 100% utilization is achieved and further increase of d does not result in an increase of the average length, since there are not any longer bad blocks to bypass. In section IV we will prove that when 100% utilization is achieved using the snake algorithm, then the expected average connection length is one. Both the partitioning and the spanning tree algorithms start well above one for small size blocks, and eventually reach one when the wafer becomes a block itself (Figure III.9). In Figure III.10a we explain why increasing the block size in a partitioning algorithm results in lower average connection length. We show the same part of a wafer structured with blocks A and with blocks B, so that the area of A is two times the area of B. Structuring the same part for the two different partitions clearly favors the larger one. The major overhead for the smaller partition is the block to block connections. Figure III.10b explain the curious zig-zag curve of the partitioning algorithm.

Our main conclusion for the average length is that the snake algorithm is the best candidate. If we want to use a partitioning algorithm then it is better to partition the wafer in bigger blocks. The opposite is true if we are interested in keeping low the worst case length. This is obviously one trade-off we have to face in our implementation, choosing the right algorithm.

We now briefly discuss the random number generator we used. We used the functions *random* and *srandom* of the *UNIX* system. Random uses a non-linear additive feedback random number generator employing a default table of size *31* long integers to return successive pseudo-random numbers in the range from *0* to *(2**31)-1*. The period of this random number generator is very large approximately *16*((2**31)-1)*. Comparing *random, srandom* with the *rand, srand* which are also *UNIX* random number generators, random produces a much more random sequence without generating a cyclic pattern. In the *UNIX Programmer's Manual, [7]* random is characterized as a

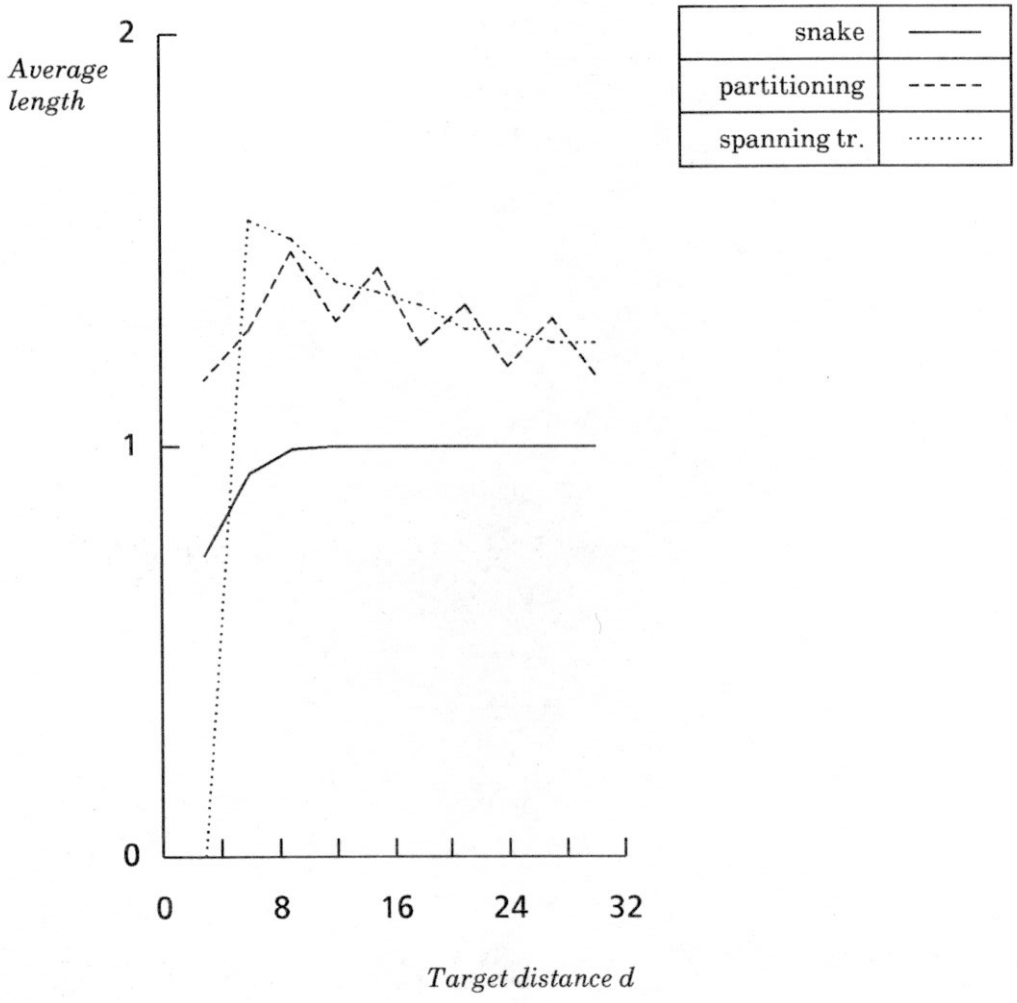


Figure III.8: Average length comparison between the snake, the partitioning and the spanning tree algorithm in relation with the worst case distance. The above results are for target distances $d=3,6,9,12,15,18,21,24,27,30$ and for wafers of size $256x256$(distances $3,9,21$) $255x255$(distances $6,12$), $252x252$(distances $15,18,24$), $250x250$(distance 27), and $253x253$(distance 30).

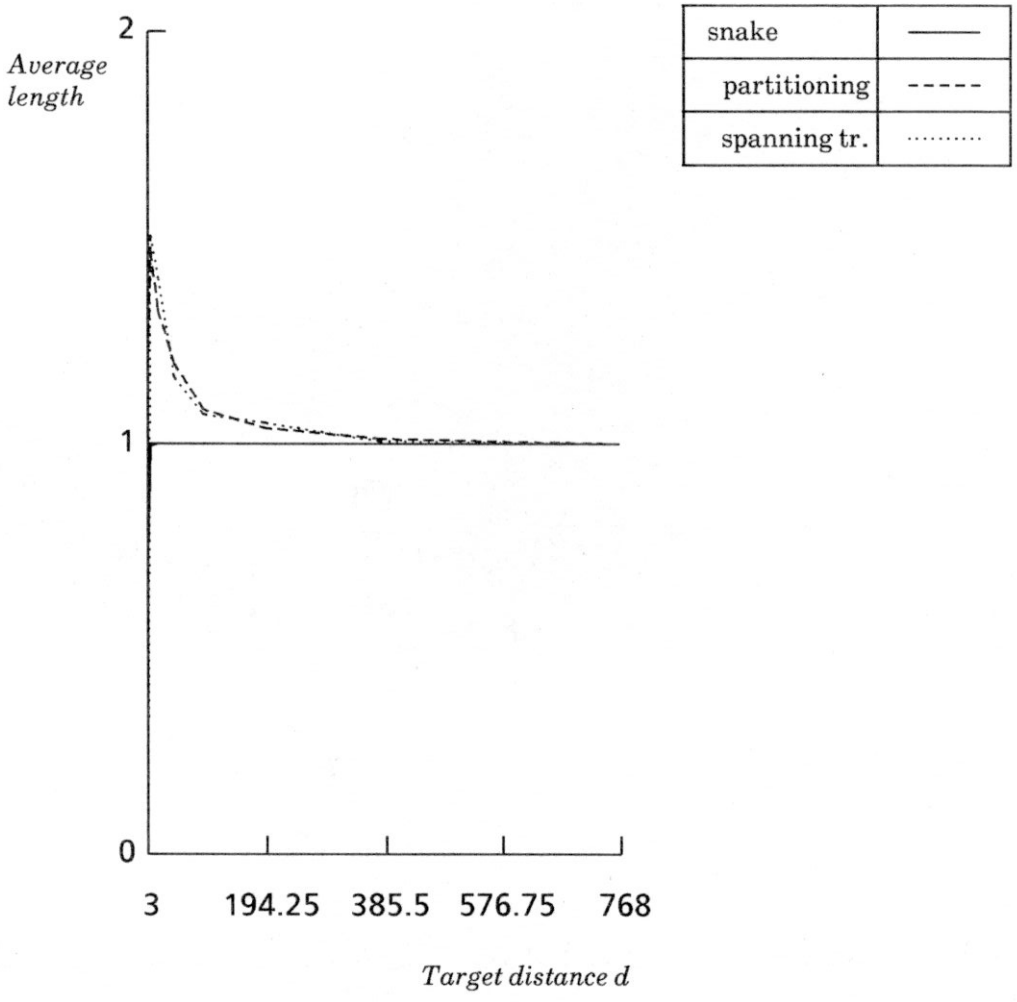
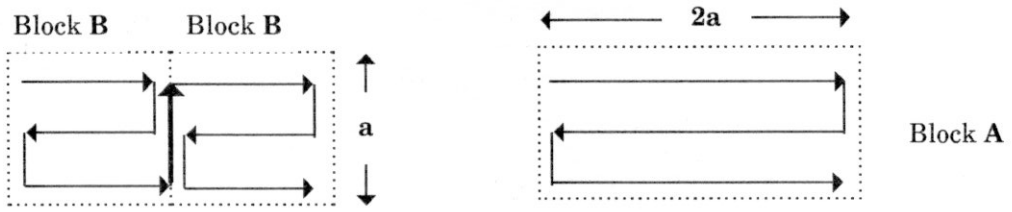


Figure III.9: Average length comparison between the snake, the partitioning and the spanning tree algorithm in relation with the worst case distance. The above results are for target distances *d*,which correspond to block sizes: *2x2,4x4,8x8,16x16,32x32,64x64, 128x128,256x256* and for wafer size *256x256*.

better random number generator. We believe that this random number generator is quite acceptable for obtaining our experimental results.

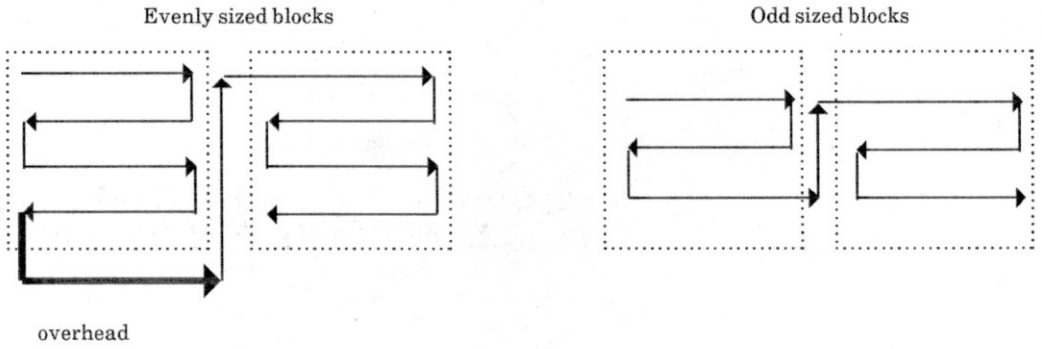


Total connection length
L=7a

Total connection length
L=6a

When we partition the wafer in **B** blocks instead of **A** we pay an extra overhead equal to the block size **a**, to connect the two **B** blocks

III.10a



III.10b

When we have a block with even sized side and we apply the snake algorithm to structure it, then the last element of the linear array inside the block will be near the left side of the block. So in order to connect it with the first live element of the next block we have to retraverse the bottom side of the block. In the case of an odd sized block, the last element of the linear array inside the block it will be near the right side of the block and it fits nicely with the first element of the next block. Because of the above overhead (in the case of the evenly sized blocks) we expect that structuring in odd sized blocks results to lower average length connections.

Figure III.10

IV. The expected utilization

In this section we analyze the expected utilization of the snake algorithm using a probabilistic model defined below. The analysis for the adaptive-snake and the partitioning algorithm are quite similar since both are snake-like algorithms. The analysis for the spanning tree algorithm has been presented in [2] and we will outline the main results.

In the following analysis we consider a 50% yield, so a cell in the wafer has *1/2* probability to be live or dead. The question of what is the maximum length required to connect all the live cells is the same as asking what is the probability of having more than a fixed number of dead cells in series to skip over. Since the event of a cell being dead is independent of the status of the other cells, if p is the probability of a cell being dead, then the probability of n consecutive cells being dead is $P = p \cdot p \ldots p = p^n$. For yield $p = 1/2$ we have that $P = (1 / 2)^n$. If we choose $n = lg_2N$, where N is the size of one side of the wafer (for simplicity we will use lgN instead of lg_2N), then $P = 1/N$. If we pick $n = 2 \cdot lgN$, then obviously $P = 1/N^2$. Since we have $NxN = N^2$ cells in the wafer, we have N^2 positions to start a bad block of $2 \cdot lgN$ dead cells (more precisely we have less than N^2 positions available : $N^2 - 2 \cdot lgN + 1$). Therefore, we do not expect to meet a bad block with length longer than $2 \cdot lgN$ and expect to meet one with length $2 \cdot lgN$.

Leighton and Leiserson [1] proved that for the linear case, the optimal length for which we expect 100% utilization is $O(sqrt (lgN))$, and they present a partitioning algorithm which achieves this bound. In our case, both the snake and the heuristic use $O (lgn)$ length to achieve 100% utilization, but the partitioning algorithm can achieve the optimal bound if we partition in blocks of sizes $O(sqrt(lgN))$ as Leighton and Leiserson did in [1] .

IV 1. The expected utilization of the snake algorithm

Our main purpose is to find a simple tool to predict our experimental results. The following analysis gives us such a tool and is based on the assumption that in each row of the wafer we can have at most one bad block. Obviously this assumption cannot hold when we expect a lot of bad blocks on the wafer. But as this number is decreased the assumption becomes more reasonable and actually gives satisfying results.

We focus our analysis on the cases where we have lengths $lgN-1$, and lgN since for these we begin to have considerable utilization. The analysis is the same for every length and we present it here for length lgN.

As we can see in section III, experimentally we found approximately the following values:

Target distance D	Expected utilization U
lgN-1	*66.5%*
lgN	*80%*

These values are almost the same for all the wafers with size larger than 64x64, but differ for smaller sizes. This happens because in the case of the small wafers the standard deviation of the average utilization is higher .

IV 1.1 A Markov chain based model

As we saw earlier we expect approximately one bad block of size $2 \cdot lgN$. To be more rigorous, we must take into consideration the boundary cells of a bad block, which are two live cells. The probability to meet a bad block of size exactly $2 \cdot lgN$ is $(1/2)^{2lgN}$ times the probability to have two live cells, one in each side of the bad block, which gives finally a probability $p=(1/2)^{2lgN+2}$. How many blocks of size exactly $2 \cdot lgN$ do we expect ? Since we have N^2 cells we expect to meet $E = p \cdot N^2$ bad blocks of size $2 \cdot lgN$. Since $E = 1 / 4$ we expect one bad block of size $2lgN$ in four samples of NxN wafers.

We can make the following table showing how many blocks of a specified length we expect to meet.

target distance D	expected number of bad blocks
1	$N^2 / 2^3 = N^2/8$
2	$N^2 / 2^4 = N^2/16$
.	
.	
lgN-2	$N^2 / 2^{lgN} = N$
lgN-1	$N^2 / 2^{lgN+1} = N/2$
lgN	$N^2 / 2^{lgN+2} = N/4$
.	
.	
$2 \cdot lgN-2$	$N^2 / 2^{2lgN} = 1$
$2 \cdot lgN-1$	$N^2 / 2^{2lgN+1} = 1/2$
$2 \cdot lgN$	$N^2 / 2^{2lgN+2} = 1/4$

As you can see from the above table, the expected number of bad blocks with size (length) bigger than $2 \cdot lgN-2$ is a number smaller than one. If we continue calculating in the same way, we can increment by one the target distance D and each time the expected number of bad blocks is divided by two, giving a geometric series. This procedure will stop when the target distance becomes N^2. We first, use the expectations to prove that the expected average connection length will be one when 100% utilization has been achieved. In the 100% utilization case, we bypass every bad block we meet to make a connection between two live cells. Since we know the number of the expected bad blocks, and each time we bypass one bad block of length L we contribute L to the average length, then one can easily find the expected average length. The only thing we need further to find is the expected number of zero length connections. In other words connections we make between live cells without having to bypass a bad block. The probability to meet two consecutive live cells is $1/2^2$. So the expected number of zero length conections is $N^2/2^2$. The expected average connection length is given by:

$$expected\ average\ length = \frac{\sum_{i=0}^{\infty} \frac{i}{2^{i+2}} N^2}{\sum_{i=0}^{\infty} \frac{N^2}{2^{i+2}}} = 2 \sum_{i=0}^{\infty} \frac{i}{2^i} = 1$$

We approximated the above sum as an infinite geometric series assumung N^2 to be large enough.

Now we can go on to analyze the expected utilization. We wish to compute the expectation of the sum of bad blocks longer than a given target distance. Assuming that the event to meet a bad block of size exactly i is independent from the event to meet a bad block of size exactly j ($i \neq j$), then the previous sum is equal to the sum of the expected number of bad blocks longer than a target distance. We approximate this sum of a finite geometric series by summing the infinite geometric series.

If the target is lgN, then during the routing of the wafer we expect to meet S bad blocks with lengths from $lgN+1$ to *infinite*, where S is given as the following sum:

$S = N^2 \cdot (1/2^{lgN+3} + 1/2^{lgN+4} + \ldots) = N^2 \cdot (1/2^{lgN+3}) / (1-1/2) = N^2 \cdot (1/4N) = N/4.$

So we expect to find $N/4$ bad blocks when the target is lgN. Since the wafer is of size NxN it is very reasonable to consider that in a uniform distribution of these bad blocks we will have at most one bad block in each row of the wafer. Based on this assumption we build the following probabilistic model. Call a row *bad* if it contains a bad block and *free* if it does not. The probability of a row to be bad for a given target distance, is equal to:

$$p = \frac{Expected\ number\ of\ bad\ blocks\ longer\ than\ the\ target\ distance}{Total\ number\ of\ rows\ N\ of\ the\ wafer}$$

The event of a row to be bad or free is independent of the status of the other rows. Now we can treat a row in the way we treated a block. To be more specific, we saw that for target distance lgN we expect $N / 4$ bad blocks. Since we assumed that a row has at most one bad block, we expect $N / 4$ bad rows. We model this by assigning the probability of having a bad row as 25%. Then, since the events are independent, the probability to have two consecutive bad rows is $p = (1/4)\cdot(1/4) = 1/16$ and generally the probability to have n consecutive bad rows is $p = (1 / 4)^n$. The probability to meet a bad row bounded by two good rows is $p = (1/4) \cdot (3/4) \cdot (3/4)$ since the probability to meet a good row (without a bad block) is $3/4$. Now, as we did for the lengths of the bad blocks, we can make the following table indicating how many times we expect to meet sets of $1,2,...,n$ bad rows.

sets of bad rows	**expected number**
1	$N \times ((9/16)\cdot(1/4))$
2	$N \times ((9/16)\cdot(1/4^2))$
.	.
.	.
$[lgN/2]-2$	$N \times ((9/16)\cdot(1/4^{[lgN/2]-2}))$
$[lgN/2]-1$	$N \times ((9/16)\cdot(1/4^{[lgN/2]-1}))$
$[lgN/2]$	$N \times ((9/16)\cdot(1/4^{[lgN/2]}))$

The probability to meet exactly $[lgN/2]$ consecutive bad rows is $((9/16)\cdot(1/4^{[lgN/2]})) = O(1/N)$, and since we have N rows we do not expect to meet more than $Nx((9/16)\cdot(1/4^{[lgN/2]})) < 1$ sets of $[lgN/2]$ consecutive bad rows. Even if we let the size of the set of bad rows to go to *infinity* (practically, to N), we do not expect to meet more than:

$S = (9N/16)\cdot(1/4^{[lgN/2]} + 1/4^{[lgN/2]+1} + ..) = (9N/16)\cdot(1/4^{[lgN/2]}) / (1-1/4) = (3N/4)\cdot(1/4^{[lgN/2]}) < 1$ sets, of $[lgN/2]$ or more consecutive bad rows. It is reasonable to say, for our simulation purposes, that we do not expect to meet a set of bad rows with size bigger than $[lgN/2]$.

Under our assumptions we know the expected number of bad sets in our wafer for the given target distance. If we could also find the expected utilization for each set of a certain size, then we would be able to find the expected utilization of the wafer. To compute the expected utilization of each set we have to compute the expected utilization of each row of the set. When we start moving on a row having a bad block, there is a probability p to meet the bad block depending on the position of the block and the direction we are moving. So with probability p we utilize the portion of the row between

the starting point and the bad block. With probability *1-p* we do not meet the bad block and we utilize the portion of the row from the starting point to the end of the row. A systematic way to explore all the possible ways to utilize certain portions of a row is using a Markov Chain model. States of the system are all the possible positions in a row. Since in the snake algorithm each row has a predetermined direction (left to right , or right to left) we also characterize each position in a row by the direction in which we are going to move. For a wafer with N elements in a row this model gives $2N$ states.

We will now consider all the possible transitions between the $2N$ states. Figure VI.1 illustrates these transitions. We have two sets of N states each, and to each set is assigned a direction. A transition between i in the first set and j in the second is characterized by the transition probability $p[i,j]$; this is the probability to go from position i in a bad row, with a fixed direction, to position j on next row with the opposite direction. In some cases this transition probability may be equal to zero.

In a bad row with direction from left to right (right to left) there are N possible positions to start a bad block, assuming that the bad blocks at the right corner (left corner) are wrapped around to the next row. If the next row is also a bad row then there may be an overlap at the right boundary (left boundary) between the two bad blocks of the consecutive bad rows. In our model we do not allow that kind of overlapping; in some cases, this leads us to a situation where we expect fewer than N starting positions for a bad block in a bad row. We included this fact in the calculated probabilities p.

If we are interested in multiple rows, the meaning of the above model is the following: If we have a set of K consecutive bad rows and we want to compute the probability $p[i,j]$ to go from position i at the first row to position j at row K' ($K' < K$), we have to explore all the possible paths of length K' from i to j. This is actually needed to compute the utilization of the K'^{th} bad row in a set of K bad rows (where i is a position at the first bad row). Furthermore, as discussed below, we are considering the block size.

We saw earlier that if the target distance is lgN then we expect $N/4$ blocks of size (length) $b > lgN$. Using these results we can also see how many blocks of an exact size we expect. If we weight each block of expected size b with the expected number of blocks of that size and take the average, we obtain something we call *the average length of a bad block*, b_{av}.For target distance lgN we have:

$$b_{av} = \frac{lgN(N^2/2^{lgN+3}) + \ldots (2lgN)(N^2/2^{2lgN+3}) + \ldots}{N^2/(2^{lgN+3}) + \ldots + N^2/(2^{2lgN+3}) \ldots} = lgN + 2$$

We want to use this length as a uniform length for all the bad blocks of size b, so that we can compute values to use as probabilities $p[i,j]$.

In Figure IV.2 we show all the possible ways to go from a position i in a row (with direction left to right) to position j in the next row (with direction right to left) if at least the first row contains a bad block of size b_{av}. The same transitions with the same transition probabilities exist for the mirror case, when we go from position $N\text{-}i+1$ in a row with direction from right to left, to position $N\text{-}j+1$ in the next row with direction from left to right.

In the case 2.a after hitting a bad block, we go down to a position j not occupied by a bad block. In the case 2.b, we see the effect of taking into consideration the block size. In calculating the utilization of the first row we do not consider the segment x as utilized because of the backtracking. This becomes more clear in case 2.c, where the utilization of the first row due to the transition (i,j) is zero.

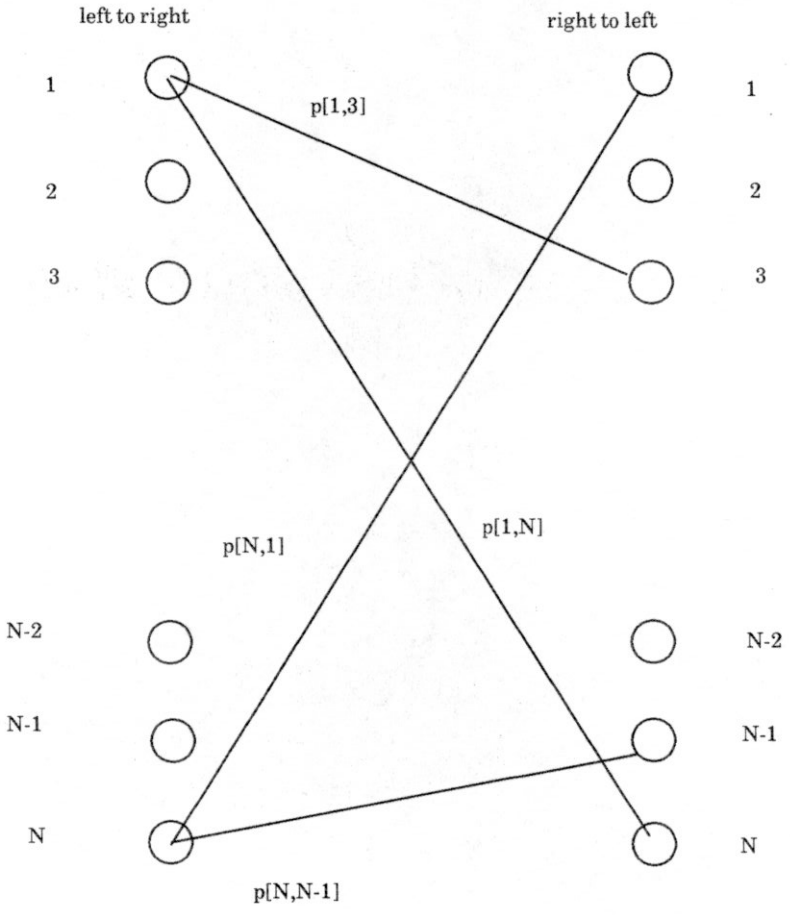


Figure IV.1 : 2N states with some transitions

The last case is when the bad block is behind the position we start moving (2.d). Then, since we assume only one bad block per row, we will not meet another bad block in the same row. If the bad block in the second row (if it contains one) does not start at position N, we will visit position N of the next row, else we will visit position N-b_{av} of the next row, since the bad block in the second row is seated between N-$b_{av}+1$ and N.

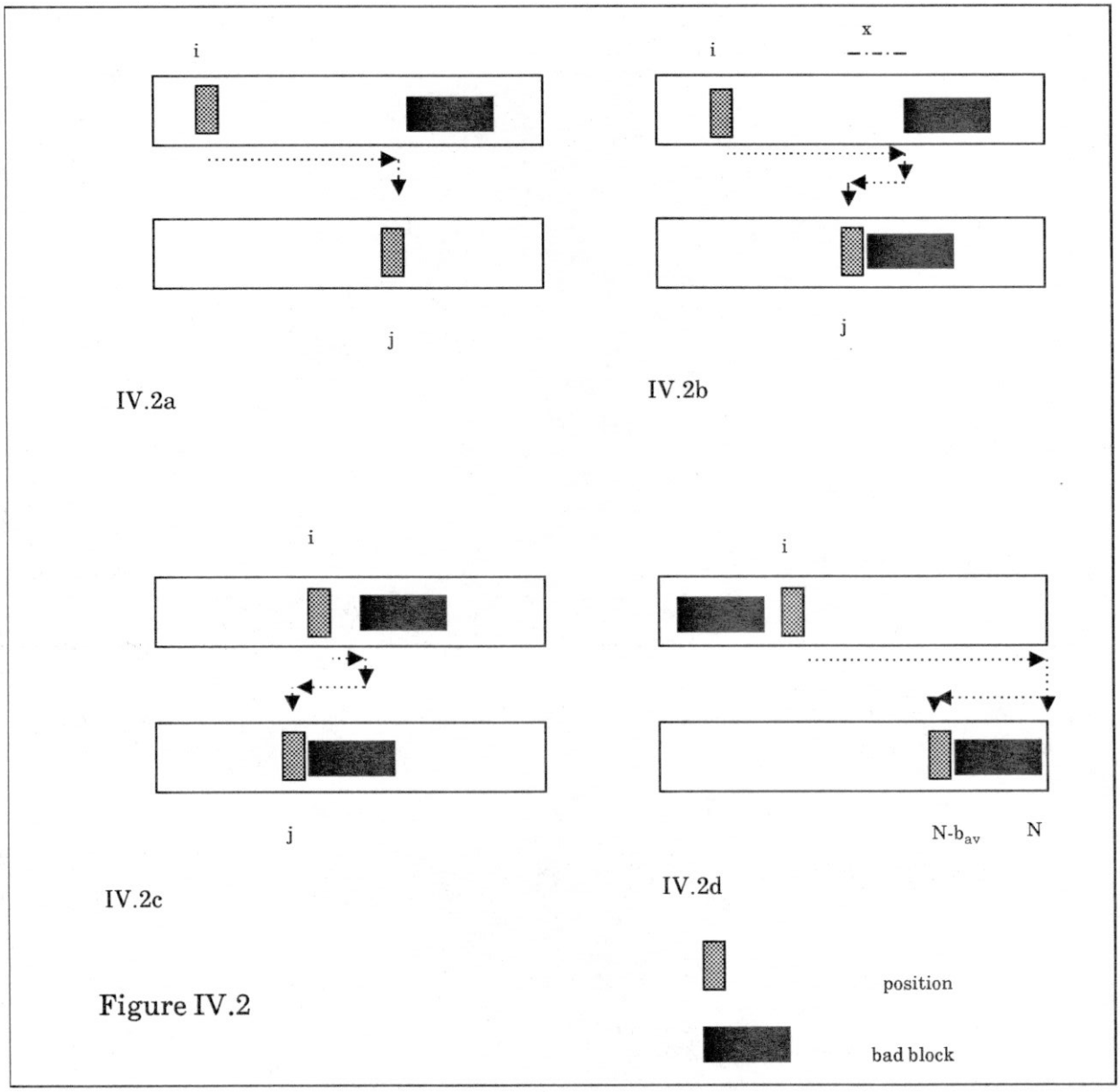


Figure IV.2

Because of the backtracking operation, the utilization of the K^{th} row in a set of K' bad rows where $K' > K$ is a little a bit less than the utilization of the same row in a set of K bad rows. We

denote as U_K the utilization of the K^{th} row for the first case, and as U_K' the utilization of the K^{th} row for the second case. We use B_K to denote the utilization for the free row after a set of K bad rows. We have as many free rows of these kinds as we have bad sets, so totally we have: $S = N/2^4 + N/2^5 + ... + 1 + 1/2 + ... = N/8$ free rows not fully utilized. Since N/4 rows contain a bad block, the remaining N/8 free rows are fully utilized. The expected utilization for the whole wafer is :

$$U = (100/N^2)[(N/4)1 + ...N/2^{lgN+3}B_{lgN+1}) + ((N/2^5 + ..N/2^{lgN+2})U_2 + ...(N/2^{lgN+3})U_{lgN}) +$$
$$(N/2^3U_1' + N/2^4U_2' + ... + N/2^{lgN+3}U'_{lgN+1})].$$

To obtain the values of U_k', U_k and B_k, we simulated the Markov chain model for wafers of size up to *64x64*. We did not simulate larger wafers because the computational time needed for larger wafers was too large. In the following table we present the results for target distance *lgN-1* computed with this model, in addition to the results for target distance *lgN*.

target distance D	**expected values of U**	**experimental values of U**
lgN-1	*62.5%*	*66.5%*
lgN	*78.5%*	*80%*

As you can see, the expected results estimated for the cases *lgN-1* and *lgN* are more pessimistic than the experimental values. The reason, as we said, it is that we considered only the case where there is only one bad block in a row. In reality, since we expect more than one bad blocks in a row, we expect more rows to be free of bad blocks. Therefore we expect an increase in the utilization of the wafer. In the case we examined, where the probability of meeting a bad block in a row is *1/4* (small), this assumption is quite reasonable and gives good results. Since the assumption may not hold, the obtained results are not a rigorous approximation, but help explain the utilization. On the other hand, for length *lgN-1*, where the same probability is *1/2*, the expected value of *U* deviates more from the experimental.

IV 2. The expected utilization of the spanning tree algorithm

The main result of the analysis in [2] is the following theorem:

THEOREM. *For arbitrarily large N and any R <p, a chain of length* $K=RN^2$ *can be connected from an NxN array with yield* $1-O(N^{-2})$ *and maximum connection length*

$$d=\left[\left(\frac{9lg((p-R)/c)}{lg(p)}\right)^{1/2}\right]$$

for some constant c >1. No more than two tracks are required in any channel.

This theorem is the final step of the analysis made in [2]. In previous lemmas they proved that if the probability q of a block to be vacant is $q<1/5$ then with high probability all the blocks of the wafer except some negligible fraction belong to a single cluster. Clearly the above probability q depends on the block size. Since we require at least four live cells in an occupied block, q is equal to:

$$q=\sum_{i=0}^{3}\binom{b}{i}(1-p)^i\,p^{b-i}$$

In our experiments for blocks of size bigger than 3x3, the probability q is less than 1/5 and 100% utilization is obtained; all the occupied blocks belong to a single cluster and we are able to utilize all of them. For blocks of size 2x2, the probability q is equal to 15/16, which is almost one; the probabilty to find a cluster of occupied blocks is almost zero. This is verified by the experimental analysis since we obtained 0% utilization. Finally, for blocks of size 3x3 the probability $q \simeq 0.25 > 1/5$, meaning that we do not expect all the occupied blocks to belong to single cluster; thus we expect to loose some of them. As you can see from the experimental results, reasonable yields are obtained for this case but we do not achieve 100% utilization.

V . Concluding remarks

In this paper, we studied four algorithms for the reconfiguration of linear regular arrays in the presence of defects. We found the expected utilization depending on the maximum allowable length of connections and we verified, for certain values of distance length, that the expected values are very close to what we found experimentally.

The snake and the adaptive-snake algorithm obtain a *100%* utilization with target distance *O(lgN)*, but the partitioning algorithm achieves the lower bound *O(sqrt(lgN))* as has been proved in [1],[2]. Partition-like algorithms (partitioning in blocks,spanning tree) are the optimum in the linear case, achieving 100% utilization at moderate interconnection distances. They have the disadvantage that the spacing of tracks between the cells is not uniform. Thus we need to know the block size in advance in order to make the tracks. Comparing the two partitioning algorithms alone, the partitioning in blocks algorithm seems superior to the spanning tree algorithm since it achieves

almost 100% utilization even for small size blocks, where the spanning tree algorithm fails to make a linear connection, for the same worst case wire length. Comparing the snake with the partitioning algorithms in terms of average length connections, the snake is clearly superior.

VI. Acknowlegements

We would like to thank Richard Lipton and Ken Steiglitz for helpful discussions. We would especially like to thank Richard Lipton for raising the issues of bus routing (Section II.5) and pointing out the doubling solution.

VII. References.

1. Broadbent, S. R. and Hammersley, J. M.: "Percolation processes", *I. Proc. Cambridge Phil. Soc.* 53(1957), pp: 629-641.

2. E. T. Cohen: RANDOM(3), *UNIX Programmer's Manual,* September 29, 1985.

3. Frisch, H., Hammersley, J.M., and Welsh, D.: "Monte Carlo estimates of percolation probabilities for various lattices". *Phys. Rev.* 126, 3(May 1, 1962), pp: 949-951.

4. J.Greene and A. Gamal: " Area and Delay penalties in Restructurable Wafer Scale Arrays ", *Journal of the ACM,* Vol 31 No 4, October 1984, pp: 694-717.

5. I. Koren: " A Reconfigurable and Fault-tolerant VLSI multiprocessor array ", in *Proc. Eighth Int. Symp. Comp. Architecture,* 1981.

6. H. T. Kung: "Why Systolic Architectures ", *Computer,* Vol. 15 , No 1, January 1982, pp: 37-46.

7. F.Leighton and C. Leiserson: " Wafer Scale Integration of Systolic Arrays ". *IEEE Transactions on Computers,* Vol C-34, No 5 May 1985.

8. J. I. Raffel and al.: "A Wafer-Scale Digital Integrator Using Restructurable VLSI", *IEEE Transactions on Electron Devices,* Vol. ED-32, No. 2, February 1985.