DIOGENES: A METHODOLOGY FOR DESIGNING FAULT-TOLERANT VLSI PROCESSOR ARRAYS

Fan R.K. Chung[*] Bell Laboratories

F. Thomson Leighton[‡] MIT

Arnold L. Rosenberg[§] Duke University

ABSTRACT. In [19], Rosenberg introduced by example the Diogenes approach to the design of fault-tolerant VLSI processor arrays. In this paper, we uncover the principles underlying the approach, and we derive from them a strategy for producing Diogenes designs for arbitrary interconnection networks. We use the strategy to derive optimal Diogenes designs of trees, grids, X-trees, and Boolean n-cubes, as well as surprisingly efficient designs of Benes permutation networks.

1. INTRODUCTION

We study here one facet of the problem of designing fault-tolerant microcircuitry, in an environment tailored to a popular VLSI architecture: arrays of identical processing elements (PEs). Our specific problem is the following.

The P(A;p,u) Problem. We want to construct an n-node array A of identical PEs. By using conservative design rules, we may assume that we can fabricate wires and switches perfectly. But we wish to design PEs aggressively, to maximize density and speed. As a result, the PEs experience debilitating faults independently, with probability p. We want to design a fault-tolerant array of PEs, that

* can be configured to simulate the array A;
* utilizes at least the fraction u of the fault-free PEs (so we fabricate n/((1-p)u) PEs to get the desired n-PE array);
* admits an efficient layout; and
* utilizes a switching mechanism that is simple (in structure and in ease of configuration).

Our specific objective is to study and extend the *Diogenes* [19] approach to the P(A;p,u) problem. The qualities of the approach -- notably, transparency to the PE designer, simplicity of configuration, and high utilization of fault-free PEs -- suggest the desirability of studying the approach with an eye to applying it to a wide variety of interconnection networks. The first fruits of our study are reported here. Related theoretical issues occupy [6].

We proceed as follows. By analyzing a sample design, we uncover the principle underlying the Diogenes approach. We use that principle to derive a strategy for producing Diogenes designs of arbitrary interconnection networks. We illustrate the strategy by deriving optimal Diogenes designs of arbitrary trees, of grids, of X-trees, and of Boolean n-cubes, as well as a surprisingly efficient Diogenes design of Benes permutation networks [3]. The paper closes with research questions awaiting resolution.

Related Work. The techniques that have been proposed in the literature for solving the P(A;p,u) problem use one of two basic strategies. The schemes in [2,9,12,13,16,22] incorporate into each PE a switching element that can connect that PE to some fixed repertoire of potential neighbors. Appropriate switch setting in the fault-free PEs interconnects some fraction of the good PEs to realize the ideal array. The schemes in [4,11,15,18,19,21] posit a switching network disjoint from the PEs. PEs are constructed as if for the ideal array, but are interconnected through the switching network rather than directly. The scheme in [10] employs a hybrid strategy.

There have been a few papers that analyze rather than present design methods. [17] evaluates four approaches for designing fault-tolerant linear arrays. His main conclusion is that such evaluations cannot be absolute: one method may be preferred when designing small arrays of large PEs, whereas another is superior for large arrays of small PEs. [14] derives a model for assessing the cost of a given design strategy. [20] presents evidence that the internal-switch strategy produces designs that consume too much layout area to be considered for any but the smallest arrays.

2. THE DIOGENES DESIGN APPROACH

2.1. An Informal Description

The major aims of the Diogenes design approach are:

* to render the design of the fault-tolerant network transparent to the designer of the PEs;
* to construct a configuration mechanism that is reconfigurable and as simple as possible to "program" to the desired structure;
* to enhance testability at a system level by building into every array a scan-in/scan-out mechanism for isolating and accessing each PE;
* to utilize (to the extent allowed by array structure) all fault-free PEs.

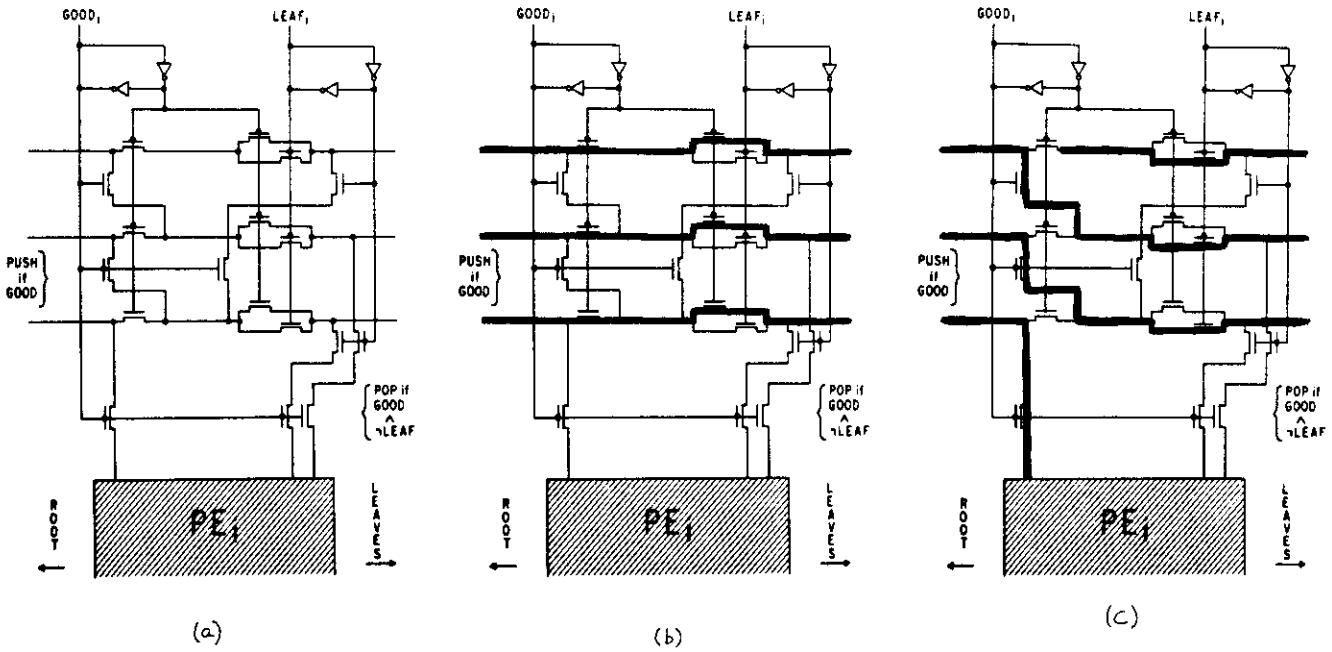
The approach can be summarized as follows.

[*]Bell Laboratories, Murray Hill, NJ 07974 USA

[‡]Mathematics Department and Laboratory for Computer Science, MIT, Cambridge, MA 02139 USA

[§]Department of Computer Science, Duke University, Durham, NC 27706 USA

0731-3071/83/0000/0026$01.00 © 1983 IEEE



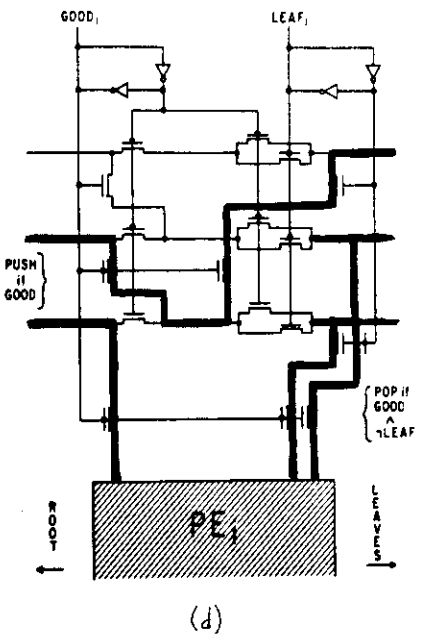


Figure 1. One cell of the Diogenes layout of the depth-3 complete binary tree: (a) unconfigured; (b) configured for a faulty PE; (c) configured for a good leaf-PE; (d) configured for a good nonleaf-PE.

One wishes to solve the P(A;p,u) problem for some given array A. One begins by fabricating $|A|/((1-p)u)$ PEs in a (logical, if not physical) line, with some number of "bundles" of wires running above the line of PEs. One then scans along the line of PEs to determine which are faulty and which are fault-free. As each good PE is encountered, it is hooked into the bundles of wires through a network of switches, thereby connecting that PE to the fault-free PEs that have already been found and preparing it for eventual connection to those that will be found. One stops looking for good PEs once $|A|$ have been found. (Alternatively, one could look for all the good PEs, and build the largest array of the given structure that one can.)

For illustration, consider an example from [19], a 7-node complete binary tree. This simple network structure needs only one bundle of wires, and that bundle needs contain only three wires.

> *Note.* To simplify exposition, we depict arrays with unit-bandwidth communication links. Removing this restriction is just a clerical matter.

One cell of a Diogenes layout of the tree appears in Fig. 1(a).

> *Note.* The lines above the PE are the single bundle needed for the layout. The switches, represented for simplicity by pass transistors, are set by two control lines, GOOD that is high when the PE is fault-free and low when it is faulty, and LEAF which is high when the PE is to act as a leaf of the tree and low otherwise. Figs. 1(b)-(d) indicate how the switches are set.

The layout of Fig. 1(a), described in terms of a depth-d tree, was derived as follows. We start out with the PEs in a line. We construct a single bundle with wires numbered 1,2,...,d. We test the PEs so that we know which are good and which are faulty. Next, we proceed down the line of PEs from right to left. As we encounter a good PE that is to be a leaf of the tree (a simple numerical formula tells us which should be leaves), we have it connect up to line 1 in the bundle (thereby preparing it to connect to its father in the tree), simultaneously having lines 1 through d-1 "shift up", to "become" lines 2 through d, respectively; switches disconnect the left parts of the lines from the right parts so that node-to-node connectivity remains correct. The bundle has thus behaved like a stack being PUSHed; see the left side of Figs. 1(c,d). When we encounter a good PE that is to be a nonleaf of the tree, we connect it to the stack/bundle in two stages. First, we have the PE connect up to lines

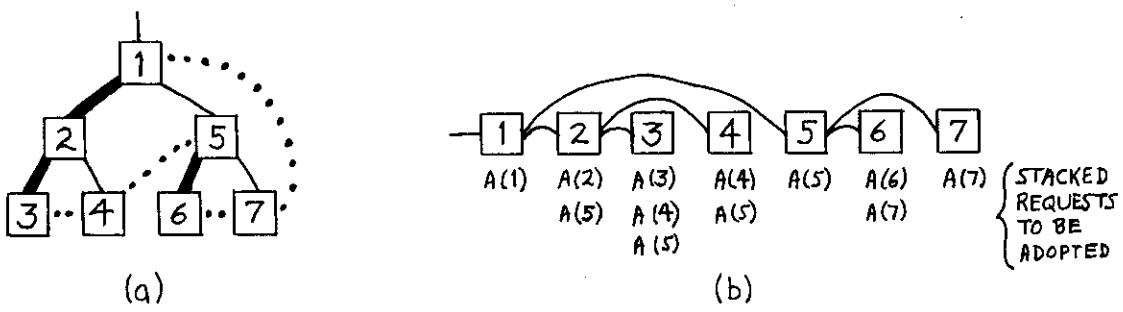


Figure 2. (a) The depth-3 complete binary tree. (b) The width-3 Diogenes (preorder) linearization of the tree.

1 and 2 of the bundle (thereby connecting the node to its sons in the tree), simultaneously having lines 3 through d "shift down" to "become" lines 1 through d-2, respectively; again switches ensure that proper node-to-node connectivity is maintained. The bundle has here behaved like a stack being POPped; see the right side of Fig. 1(d). Second, we have the PE PUSH a connection onto the stack, to prepare for eventual connection to its father in the tree. The process we have described here lays the tree out in ***preorder*** (cf. Fig. 2). Hence, a d-wire stack/bundle suffices to lay out a depth-d complete binary tree.

> *Note.* Our design strategy will require highlighting certain edges of the network being laid out, as well as adding new edges to it. Highlighted edges in Figs. 2-4 are represented by bold lines; added edges are represented by dotted lines. The significance of both kinds of special edges will be explained in Section 4.

The preceding example should suffice to introduce the Diogenes approach. The designs in [19] simplify the problem of configuring the network by organizing their wire bundles as either stacks (as our example) or queues. For example, two bits of information (= control lines) per PE suffice to configure a line of PEs into any-depth complete binary tree: one bit tells whether or not a PE is good; the other tells whether or not it's a leaf. A less structured bundle (e.g., a crossbar) would require a number of bits per PE proportional to the depth of the tree.

2.2. Stack-Induced Layouts

The Diogenes design approach is distinguished from other external-switch approaches (e.g., [4,18,21]) in its structuring switches so that wire bundles behave as stacks or queues. It is this organizing principle that we exploit to extend the approach to arbitrary interconnection networks.

We restrict attention here to Diogenes designs that organize wire bundles as stacks. Stack-bundles are (as pointed out in [19]) easier to implement than queue-bundles, thereby reinforcing our quest for easily applicable results. Moreover, it is our experience that learning to reason about stacks helps one to reason about queues.

Finally, an avenue for simplification: The Diogenes "recipe" has two parts: a faulty PE is passed by without hooking it into any stack/bundle; a fault-free PE is hooked into the bundles in some relatively complicated way. The former prescription is not interesting: one ignores bad PEs by straightforward use of the GOOD control lines that appear in every Diogenes design (cf. Fig. 1). The interesting aspect of Diogenes designs is how they use stacks to realize interconnections among the good PEs. Recognizing this, we simplify our study by ignoring the GOOD lines and their role in network configuration. This relegates to the background the fault-tolerating aspect of the motivating problem and concentrates solely on the problem of how to use stacks to configure a line of (fault-free) PEs into any desired array structure.

The essence of having a wire-bundle act as a stack is that inter-PE connections made using that bundle never cross. (This is both necessary and sufficient.) Our topic of study thus reduces to the following. As is customary, we view arrays as undirected graphs (cf. [12]).

The Formal Layout Problem. To partition the edges of the graph G and to lay G out in the plane in such a way that:
* the vertices of G lie on a line;
* all edges of G lie above the line;
* no two edges in the same block of the partition cross.

In view of our earlier remark, it is clear that our problem of realizing arrays using stacks is equivalent to the formal problem just stated. A third formulation will be useful for insight.

The graph G is ***outerplanar*** if its vertices can be placed on a circle in such a way that the edges of G are noncrossing chords of the circle.

Proposition 1. [6] A graph is realizable with one stack if, and only if, it is outerplanar.

> We are, thus, studying ***multi-outerplanar*** graphs: The graph G is ***k-outerplanar*** if it is the union of k outerplanar graphs whose outerplanarity is witnessed by the same layout of Vertices(G) on a circle.

Proposition 2. [6] A graph is realizable with k stacks if, and only if, it is k-outerplanar.

2.3. The Quality of a Diogenes Layout

Three parameters measure the quality of a Diogenes layout of a graph G:
1. the number of stacks employed in the layout;
2. the (a) ***individual*** and (b) ***cumulative (stack)widths*** (= number of lines) of the stacks used in the layout;
3. the number of control bits needed to configure the layout: given the layout, each vertex v of G has an associated vector of pairs of nonnegative integers, called its ***type***,

$T(v) = \langle\langle L_1,R_1\rangle, \langle L_2,R_2\rangle, \ldots\rangle$:
each L_i (resp., R_i) is the number of edges incident to v that connect via Stack i to vertices lying to the left (resp., right) of v. This measure is the base-2 logarithm of the number of distinct vertex-types in the layout of G, i.e., the number of "control" bits needed to configure the layout in the presence of faults.

We weight these measures in decreasing order of importance when "optimizing" layouts. In [6] we study tradeoffs among them.

3. DIOGENES LAYOUTS OF TREES

Our layout of the n-node complete binary tree is optimal with respect to all three quality measures: its one stack respects the outerplanarity of the tree (Prop. 1); its log n width respects the lower bound of [5]; and its two control bits per PE respects our insistence on fault tolerance. We can do almost as well with arbitrary trees.

Fact 1. (a) Any n-node k-ary tree admits a Diogenes layout with one stack of width $\leq W(n) =_{def} k/2 \log n$.
(b) There is a fixed layout using a single width-W(n) stack and using $1+2\log_2(k+1)$ control bits per PE, that can be configured to any k-ary tree having n or fewer nodes.

Proof Sketch. Let G be a graph. One *adds a fringe* to a vertex v of G by appending to v a line of (possibly 0) vertices:

$$v\text{–}v_1\text{–}v_2\text{–}\ldots\text{–}v_r \quad r\geq 0.$$

A *fringing* of G is a graph obtained by adding a fringe to each vertex of G.

Concentrate on one vertex v of G. Say that when G is laid out, v is flanked by vertices u and w. Let v have two fringes, $v_1, \ldots, v_r$ and $v'_1, \ldots, v'_s$ (one or both can be empty). Lay the fringes out in the indicated order, between either u and v or v and w. To choose the sides and stacks, look at v's type. Put the first fringe on the side and the stack having the smallest integer entry in v's type; place the second fringe using the smallest entry in v's (now altered) type. This increases the cumulative stackwidth by at most 1, while leaving the stacknumber unchanged.

Fact 1 now follows by verifying that any k-ary tree T can be "built" by levels, by starting with a single vertex and "double"-fringing the graph $\leq k/2 \log |T|$ times. The number of control bits follows from counting the number of distinct vertex-types when all vertices have degree at most k+1. []

4. A GENERAL LAYOUT HEURISTIC

The layout technique of Fact 1 builds explicitly on the structure of the graphs being laid out. It would be useful to know what to look for in an interconnection network's structure to help one find efficient layouts of arrays of that structure. Experience from numerous Diogenes layouts has led us to the following heuristic.

The graph G is *hamiltonian* if there is a cycle in the graph that meets each vertex just once. The graph G' is an *augmentation* of the graph G if G' is obtained by adding $k \geq 0$ edges to G.

4.1. A Heuristic Layout Procedure.

To find a Diogenes layout for G:

1. Augment G (if necessary) so that it has a hamiltonian cycle.
2. Cut the cycle to obtain a layout of G in a line.
3. Assign edges to stacks using edge coloring as in [8].

As one example of the heuristic, our layout of complete binary trees results from applying the procedure to "preorder" augmentations of the trees. (See Fig. 2(a); the chosen cycle consists of bold and dotted lines.) Other one-stack layouts of trees exist (cf. [7]), but none has smaller width or number of control bits.

4.2. Origin of the Heuristic

The heuristic had two origins. First, the heuristic embodies the proof of Proposition 1. Second, it embodies the proof of the following generalization of Proposition 1 to a wide class of planar graphs.

A graph is *subhamiltonian* if it has a planar hamiltonian augmentation.

Proposition 3. [6] The graph G is two-stack realizable (= 2-outerplanar) if, and only if, it is subhamiltonian.

4.3. Applications of the Heuristic.

Square Grids

The augmented cycle formed by row-by-row sweeps in a square grid, as indicated in Fig. 3(a), leads to the layout of Fig. 3(b), which is optimal in number of stacks (the grid is planar but not outerplanar), stackwidth (see, e.g., [18]), and number of node types (the layout distinguishes only between east-to-west and west-to-east rows of the grid).

Fact 2. The nxn square grid admits a two-stack Diogenes layout with stackwidth n and with two node types. This realization is optimal in all three parameters.

X-Trees

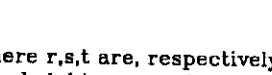
The *depth-d X-tree* X(d) is the augmentation of the depth-d complete binary tree that adds edges going across each level of the tree; see Fig. 4(a).

X(d) has cutwidth d and is subhamiltonian, but not outerplanar. Thus the best possible Diogenes layout would use two stacks of width d. It is very hard to find a two-stack layout of stackwidth smaller than roughly 2^d. (All obvious hamiltonian cycles lead to this enormous width.) The hamiltonian augmentation of X(d) of Fig. 4(a) leads to the stackwidth-3d two-stack layout of Fig. 4(b).

Fact 3. X(d) admits a Diogenes layout with two stacks, one of width 2d and one of width 3d. This realization is optimal in stacknumber and within a factor of 5 of optimal in stackwidth.

The only subtlety here is to verify the claimed stackwidths. As part of our sketched verification, we describe the layout more formally. We proceed by induction. Say that we have a layout of X(d-1) with the claimed parameters and the following form. We depict the layout schematically by its linearization of the vertices, together with a few relevant edges. For simplicity we draw edges in stack 1 above the line, those in stack 2 below the line.

Layout 1.



here r,s,t are, respectively, the root of X(d-1) and its left and right sons; α,β are the strings comprising the rest of the trees' vertices. Assume for induction that in Layout 1: (1) the left spine nodes [= leftmost nodes at each level] of X(d-1) appear in leaf-to-root order in α; the right spine nodes [the rightmost nodes at each level

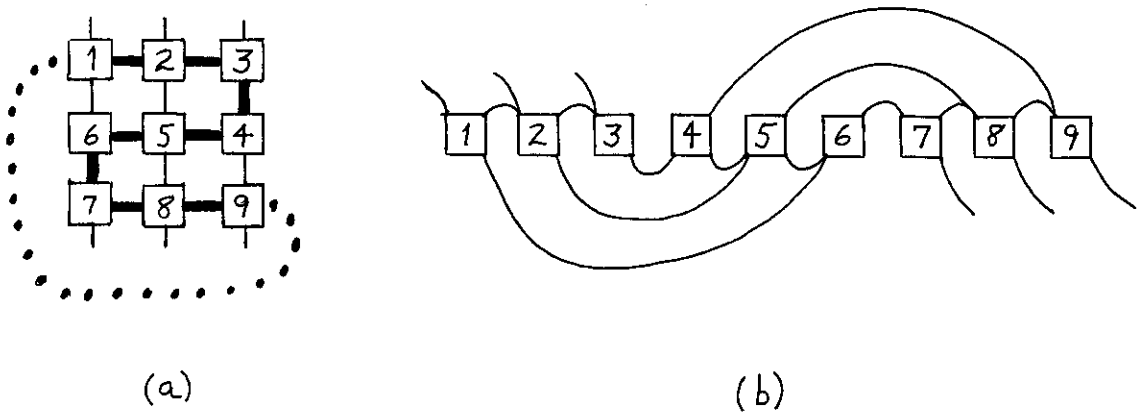
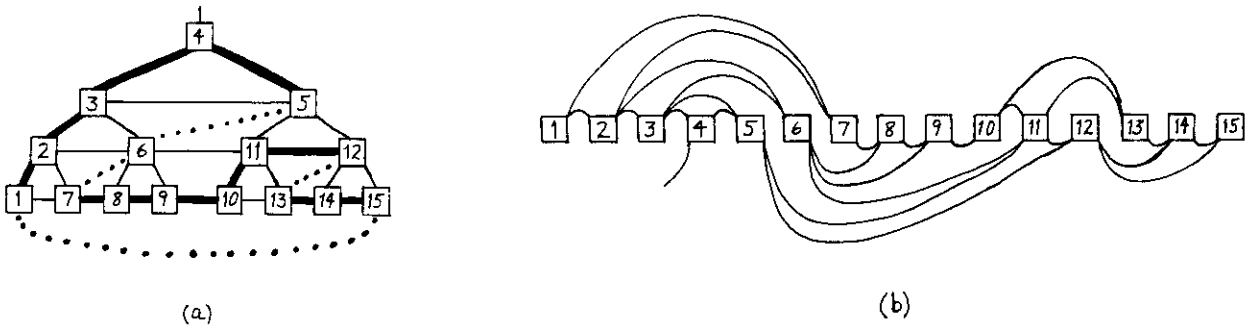


Figure 3. The square grid and its Diogenes linearization.

Figure 4. (a) The depth-4 X-tree X(4). (b) A two-stack, width-4 Diogenes linearization of X(4).



appear (nonconsecutively) in root-to-leaf order in β; (2) the nodes r,s,t, and all of the left and right spine nodes are *exposed* from the bottom, in the sense that no edges pass totally under them; (3) the width of stack 1 is $\leq$ 2d-2; (4) the width of stack 2 is 0 below the left spine nodes and is < 3k-3 to the right of the level-(d-k-1) spine nodes. Take a second copy of Layout 1:

Layout 2. 

The layout of X(d) [whose vertex-set is the union of the vertex-sets of its two depth-(d-1) sub-X-trees, plus a root node r*] is obtained from the indicated layouts as follows:

Layout 3. 

A careful analysis of the composite layout extends the induction. Analysis of small trees completes the induction, which establishes our claims.

Benes Permutation Networks

Let n be a power of 2. The *n-input Benes network* B(n) is defined inductively as follows; see Fig. 5(a).

* B(2) is the complete bipartite graph $K_{2,2}$ on two *input* vertices $i_{1,1}$, $i_{1,2}$ and two *output* vertices $o_{1,1}$, $o_{1,2}$.
* B(n) is obtained from two copies of B(n/2); n new input vertices, $i_{n,1},i_{n,2},\ldots,i_{n,n}$; and n new output vertices, $o_{n,1},o_{n,2},\ldots,o_{n,n}$. For each $1\leq k\leq n$, one adds edges creating one copy of $K_{2,2}$ with "inputs" $i_{n,k}$ and $i_{n,k+n/2}$ and "outputs" $i_{n/2,k}$ and $i'_{n/2,k}$ and one copy of $K_{2,2}$ with "inputs" $o_{n/2,k}$ and $o'_{n/2,k}$ and "outputs" $o_{n,k}$ and $o_{n,k+n/2}$ (primed vertices come from the second copy of B(n/2)).

Benes networks are nonplanar, hence require at least three stacks. We have not yet achieved this bound, but we have found a six-stack realization, by means of the hamiltonian cycle that alternates running up and down the "columns" of inputs and outputs of B(n); see Fig. 5(a). We use three stacks to connect each "column" of vertices to the next; and we alternate sets of three stacks as we proceeed along the graph. It is surprising that any family of graphs capable of realizing all permutations can be laid out with a fixed number of stacks.

Fact 4. B(n) admits a Diogenes layout using six stacks, each of width n. This realization is within a factor of 2 of optimal in stacknumber and within a factor of 6 in stackwidth.

The same layout strategy yields layouts of comparable efficiency for structural relatives of B(n), including (log n)-stage cyclic shifters.

The Boolean n-Cube

The *Boolean n-cube* C(n) has as vertices the set of all length-n binary strings. String-vertices are adjacent just when they have unit Hamming distance. Thus C(n) has 2^n vertices and $n2^n$ edges. Since C(n) is hard to visualize for n>3, we describe its efficient layout in terms of strings rather than the graphical medium of hamiltonian cycles.

Fact 5. C(n) is n-stack realizable, with one stack of width 2^i for each $0\leq i<n$. This realization is within a factor of 2 of optimal in both stacknumber and cumulative

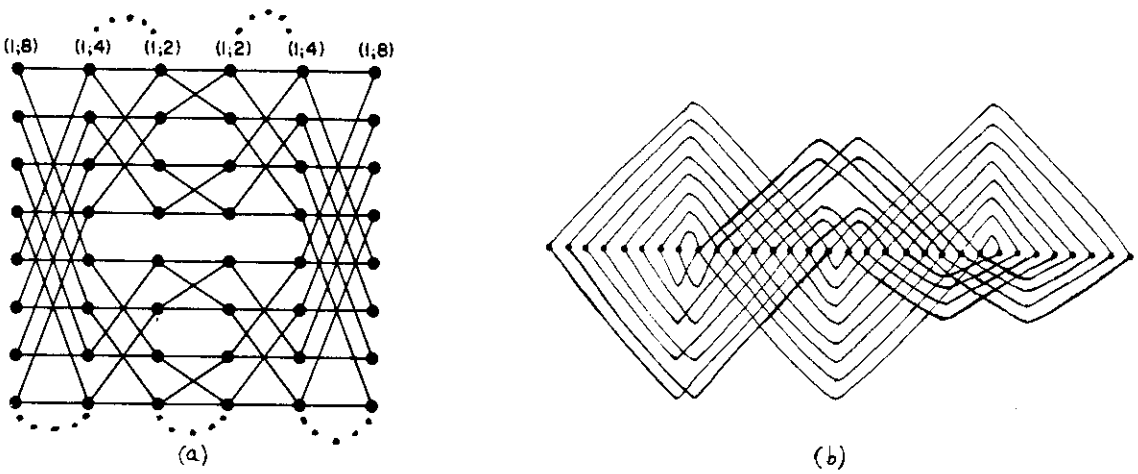


Figure 5. (a) The 8-input Benes network. (b) A six-stack layout of the first three levels of the network.

stackwidth.

The lower bound on stacknumber is immediate from three facts: (a) Stacknumber(C(n)) ≥ the number of outerplanar graphs into which C(n) can be decomposed; (b) an N-vertex outerplanar graph has at most 2N edges; (c) C(n) has $n2^n = N \log_2 N$ edges. The lower bound on cumulative stackwidth is easy to derive.

The upper bound is seen most easily by describing inductively the linearization of C(n)'s vertices.

* C(1)'s vertices are laid out as follows.

  0 1

  so one width-1 stack suffices.
* Assume that C(n) is realized with n stacks of widths 1,2,...,2^{n-1}, via the linearization (letting $N=2^n$)

  $$\beta_1\beta_2\cdots\beta_N$$

  each β_i being a distinct length-n binary word. The following layout for C(n+1):

  $$0\beta_1 0\beta_2\cdots 0\beta_N 1\beta_N\cdots 1\beta_2 1\beta_1$$

  uses just one more stack, of width $N=2^n$. This extends the induction.

5. OPEN PROBLEMS

1. Is there a fixed number S such that all planar graphs are S-stack realizable?
2. Is there a fixed number S such that all N-node outerplanar graphs can be realized with S stacks of width proportional to log N?

   The *depth-n ladder* L(n) is an nx2 grid. (a) L(n) is outerplanar, hence one-stack realizable. (b) Any one-stack realization of L(n) has stackwidth ≥ n/2. (c) There is a two-stack unit-width realization of L(n).
3. Is there a fixed number S such that all N-node planar subhamiltonian graphs can be realized with S stacks of width proportional to $N^{1/2}$?
4. Can Benes networks be realized with fewer than six stacks?

ACKNOWLEDGMENTS. The research of the second author (FTL) was supported in part by: a Bantrell fellowship, DARPA Contract N00014-80-C-0622, and Air Force Contract OSR-82-0326. The research of the third author (ALR) was supported in part by NSF Grant MCS-8116522.

The authors would like to thank Gary Miller for helpful and stimulating conversations on this topic.

REFERENCES

1. D.P. Agrawal (1982): Testing and fault tolerance of multistage interconnection networks. *Computer 15*, 41-53.
2. R.C. Aubusson and I. Catt (1978): Wafer-scale integration -- a fault-tolerant procedure. *IEEE J. Solid-State Circuits, SC-13*, 339-344.
3. V.E. Benes (1964): Optimal rearrangeable multistage connecting networks. *Bell Syst. Tech. J. 43*, 1641-1656.
4. S.N. Bhatt and C.E. Leiserson (1982): How to assemble tree machines. *Proc. 14th ACM Symp. on Theory of Computing*, 77-84.
5. R.P. Brent and H.T. Kung (1980): On the area of binary tree layouts. *Inf. Proc. Let. 11*, 44-46.
6. F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1983): Multi-outerplanar graphs. In preparation.
7. D. Dolev and H. Trickey (1982): Embedding a tree on a line. IBM Report RJ-3368.
8. S. Even, A. Pnueli, A. Lempel (1972): Permutation graphs and transitive graphs. *J. ACM 19*, 400-410.
9. D. Fussell and P. Varman (1982): Fault-tolerant wafer-scale architectures for VLSI. *Proc. 9th Int'l Symp. on Computer Architecture.*
10. D. Gordon, I. Koren, G.M. Silberman (1982): Embedding tree structures in fault-tolerant VLSI hexagonal grids. typescript.
11. J.W. Greene and A. El Gamal (1982): Area and delay penalties in restructurable wafer-scale arrays. *Proc. 3rd Caltech Conf. on VLSI.*
12. J.P. Hayes (1976): A graph model for fault-tolerant computing systems. *IEEE Trans. Comp., C-25*, 875-884.
13. I. Koren (1981): A reconfigurable and fault-tolerant VLSI multiprocessor array. *Proc. 8th Int'l Symp. on Computer Architecture*, 425-442.
14. I. Koren and M.A. Breuer (1982): On area and yield considerations for fault-tolerant VLSI processor arrays. USC Digital Integrated Systems Center Rpt. DISC/82-5.
15. F.T. Leighton and C.E. Leiserson (1982): Wafer-scale integration of systolic arrays. *Proc. 23rd IEEE Symp. on Foundations of Computer Science*, 297-311.
16. F. B. Manning (1977): An approach to highly integrated, computer-maintained cellular array. *IEEE Trans. Comp., C-26*, 536-552.
17. C.D. Rogers (1982): A critical survey of four techniques for the construction of fault-tolerant multiprocessor arrays. In Duke Tech. Rpt. CS-1982-17.
18. A.L. Rosenberg (1981): Routing with permuters: Toward reconfigurable and fault-tolerant networks. Duke Tech. Rpt. CS-1981-13.
19. A.L. Rosenberg (1982): The Diogenes approach to testable fault-tolerant arrays of processors. *IEEE Trans. Comp.*, to appear; Duke Tech. Rpt. CS-1982-6.
20. A.L. Rosenberg (1982): On designing fault-tolerant arrays of processors. Duke Tech. Rpt. CS-82-14; submitted for publication.
21. L. Snyder (1981): Overview of the CHiP computer. In *VLSI 81: Very Large Scale Integration* (ed. J. P. Gray) Academic Press, London, pp. 237-246.
22. W.W. Wong and C.K. Wong (1982): Minimum k-hamiltonian graphs. IBM Report RC-9254.