A Cost-Effective Streamlining of the DIOGENES Design Methodology

Marc Picquendar & Arnold L. Rosenberg

Computer Science Technical Report 93-47
May, 1993

A Cost-Effective Streamlining of the DIOGENES Design Methodology*

Marc Picquendar and *Arnold L. Rosenberg*
May 7, 1993

Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003, USA

Abstract. The original DIOGENES design methodology achieves its fault-tolerant layouts of VLSI processor arrays by designing the network that interconnects the processors as a (possibly large) number of bundles of wires, where each bundle is organized as either a *stack* or a *queue*. The benefits of the methodology sometimes come only at high cost, in terms of both the needed configuration hardware and the algorithmic cost of finding efficient configurations. In this paper, we present an improved version of the original methodology, that simultaneously streamlines the design process and produces more cost-effective layouts. The fundamental change imposed by this new methodology replaces the possibly large number of "stacks" and/or "queues" of the original methodology by a single *generalized insertion queue* (GIQ for short), independent of the topology of the network. The new methodology has three major benefits:

1. Whereas the problem of finding an efficient configuration for an array is hard for the original methodology (in fact, NP-complete for the "stack" version), *the new methodology admits a configuration algorithm that operates in almost linear time.*
2. *The hardware needed to realize a* GIQ *layout of an array is usually dramatically less than that needed to realize a "stack" or "queue" layout:* while the individual switches for a GIQ are somewhat larger than those for a stack or queue, a layout based on GIQs requires only a single, small bundle, and thus simpler control hardware, in contrast to the possibly many, possibly large bundles of the original methodology.
3. Once an assignment of edges to "stacks" or "queues" in the original methodology has been chosen, the array is committed to a restricted set of topologies; by contrast,*a* GIQ *lay out can realize any linearization of any graph of given cutwidth.* One can use this flexibility to lay out dynamically reconfigurable arrays of processors.

* This research was supported in part by NSF Grant CCR-90-13184.

1 Introduction

1.1 The DIOGENES Design Methodology

Philosophy. The DIOGENES methodology [12, 4] for designing fault-tolerant arrays of identical processing elements (PE) is one of a class of methodologies that achieve fault tolerance by interconnecting PEs through a reconfigurable interconnection network [13]. All of these methodologies view a processor array as a graph whose nodes are the PEs of the array and whose edges are (bi-directional) links interconnecting the PEs. The uniqueness of the DIOGENES methodology lies in its design of the interconnection network. Specifically, the methodology views the array's PEs as laid out in a (logical) line, with a number of *bundles* of wires running above the line. Each bundle is organized to behave as either a *stack* of wires or as a *queue* of wires, in a sense made explicit in [12] and described below.

Stacks and queues of wires. Consider a bundle of n data lines, numbered $1, 2, \ldots, n$, that carry the data between PEs in a processor array. The first consequence of the bundle's observing either a stack or queue discipline is that, if only $k \leq n$ of the lines in the bundle are carrying "active" data, then those k lines are the ones numbered $1, 2, \ldots, k$. The other consequences are as follows.

Assume first that the bundle observes a *stack discipline*. When the bundle passes from left to right past a PE that needs the data from $m \leq k$ of the active data lines, then the PE performs m POP operations on the bundle. Each POP consists of routing line 1 of the bundle into the PE and, simultaneously, routing each line i of the bundle $(1 < i \leq n)$ onto line $i - 1$. When the bundle passes from left to right past a PE that needs to insert $l \leq n - k$ data lines, then the PE performs l PUSH operations on the bundle. Each PUSH consists of routing a line from the PE onto line 1 of the bundle and, simultaneously, routing each line i of the bundle $(1 \leq i < n)$ onto line $i + 1$.

Assume next that the bundle observes a *queue discipline*. When the bundle passes from left to right past a PE that needs the data from $m \leq k$ of the active data lines, then the PE performs m DEQUEUE operations on the bundle. Each DEQUEUE consists of routing line 1 of the bundle into the PE and, simultaneously, routing each line i of the bundle $(1 < i \leq n)$ onto line $i - 1$. When the bundle passes from left to right past a PE that needs to insert $l \leq n - k$ data lines, then the PE performs l ENQUEUE operations on the bundle. Each ENQUEUE consists of routing a line from the PE onto line $m + 1$ of the bundle, where m is the number of active data lines at the instant of the ENQUEUE operation. Of course, in a properly designed circuit, one never tries to "augment" a "full" stack or queue, nor to "extract" from an "empty" one.

Benefits. One can view the main benefit of the DIOGENES methodology and its unique network organization as residing in the fact that the entire design and implementation process is amenable to rigorous mathematical verification and analysis. There is a sizable literature on the logical linearization of PEs: when edges of the network are to be laid out via stacks, the linearization problem is equivalent to the well-studied problem of *embedding graphs in books*; when edges of the network are to be laid out via queues, the linearization problem has received less attention but is still well understood [8], [9]. The problem of configuring the interconnection network to achieve a desired network topology efficiently has been studied in depth [10]. Even some of the "engineering" aspects of making the final implementation efficient have received attention [1], [3], [11].

Costs. The benefits of the methodology do not come without cost, both in hardware and algorithmic resources.

1. DIOGENES designs of all but the (structurally) simplest arrays require several stacks or queues of wires [2], [5], [9], hence considerable circuitry.

2. DIOGENES designs that attempt to minimize the number of stacks or queues of wires often do so only at the cost of dramatically increased overall cutwidth[1] of the layout [5], [9], [14].

3. The general problem of finding an efficient assignment of array links to wire bundles is algorithmically difficult (technically, NP-complete) for both stack-based and queue-based layouts. For stack-based layouts, the NP-completeness persists even when one is given the desired linearization of the array's PEs *a priori* [5].

1.2 A New Version of the Methodology

Generalized Insertion Queues. Our new version of the DIOGENES methodology is based on a modification of the queue data structure, which we call the *generalized insertion queue* or GIQ. A GIQ is a linear list $\langle \delta_1, \ldots, \delta_p \rangle$. Position 1 is the *head* and position p the *tail* of the GIQ; p is the *population* of the GIQ. The GIQ data structure supports the following operations.

- The REMOVE operation extracts the element at the head of the GIQ and moves all remaining elements forward one position, thereby decreasing the population of the GIQ by one:

[1]The *cutwidth of a linear layout of a graph $\mathcal{G}$* is the largest number of edges that cross above an inter-node gap in the layout. The *cutwidth of $\mathcal{G}$* is the smallest cutwidth of any linear layout of $\mathcal{G}$.

$$\text{REMOVE} : \langle \delta_1, \ldots, \delta_p \rangle \mapsto \delta_1, \langle \delta_2, \ldots, \delta_p \rangle$$

- The INSERT-k operation adds one element to a GIQ by placing it in position k, increasing the population of the GIQ by 1. Elements at positions 1 to $k-1$ remain in their positions, while elements at positions i ($i \geq k$) are moved to the next higher position:

$$\text{INSERT-}k : \delta, \langle \delta_1, \ldots, \delta_p \rangle \mapsto \langle \delta_1, \ldots, \delta_{k-1}, \delta, \delta_k, \ldots, \delta_p \rangle$$

It is clear that GIQ operations can emulate stack and queue operations: The REMOVE operation on a GIQ has the same effect as a POP operation on a stack, or a DEQUEUE operation on a queue. An INSERT-1 operation on a GIQ has the same effect as a PUSH operation on a stack. An INSERT-$(p+1)$ on a GIQ with population p has the same effect as an ENQUEUE operation on a queue that holds p items. A GIQ can thus behave alternatively as a stack or a queue. In fact, the choice of insertion position allows one to fix the order of removal of elements, independently of the order of insertion. This added flexibility greatly simplifies the layout of graphs using GIQs, as will be made clear in the rest of this section.

Sample Layouts. We first illustrate the differences between stack, queue and GIQ layouts by comparing three layouts of the graph of Figure 1. As pointed out in [9], certain configurations of edges constitute obstacles to minimizing the number of queues or the number of stacks for a fixed-order layout of a graph. Specifically, a linearized graph containing a *k-rainbow* (i.e., a set k fully nested edges) can be laid out using k queues, but not less. On the other hand, a linearized graph containing a *k-twist* (i.e., a set of k fully intersecting edges) can be laid out with k stacks, but not less.

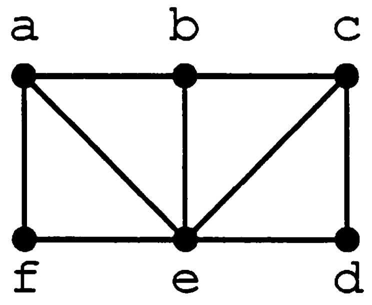

Figure 1: Example graph

Stack layout. The linearization (d, e, f, a, b, c) (see Figure 2) does not contain any intersection and can therefore be laid out using a single stack. On the other hand edges (d, c), (e, b) and (f, a) form a 3-rainbow, so 3 queues would be required for a queue layout. We show below the sequence of PUSH and POP operations on the stack. In conformance with the stack regimen, edges are POPed in the reverse order in which they are PUSHed.

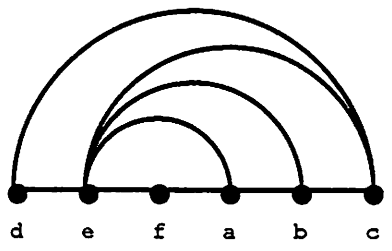

Figure 2: One-stack layout

Node	Operation	Composition of the stack
d	PUSH (d,c)	(d,c)
	PUSH (d,e)	$(d,e),(d,c)$
e	POP (d,e)	(d,c)
	PUSH (e,c)	$(e,c),(d,c)$
	PUSH (e,b)	$(e,b),(e,c),(d,c)$
	PUSH (e,a)	$(e,a),(e,b),(e,c),(d,c)$
	PUSH (e,f)	$(e,f),(e,a),(e,b),(e,c),(d,c)$
f	POP (e,f)	$(e,a),(e,b),(e,c),(d,c)$
	PUSH (f,a)	$(f,a),(e,a),(e,b),(e,c),(d,c)$
a	POP (f,a)	$(e,a),(e,b),(e,c),(d,c)$
	POP (e,a)	$(e,b),(e,c),(d,c)$
	PUSH (a,b)	$(a,b),(e,b),(e,c),(d,c)$
b	POP (a,b)	$(e,b),(e,c),(d,c)$
	POP (e,b)	$(e,c),(d,c)$
	PUSH (b,c)	$(b,c),(e,c),(d,c)$
c	POP (b,c)	$(e,c),(d,c)$
	POP (e,c)	(d,c)
	POP (d,c)	-

Table 1: Stack Layout

Queue layout. The linearization (d, a, e, b, f, c) (see Figure 3) does not contain any rainbow and can therefore be laid out using a single queue. On the other hand, it cannot be laid out using fewer than 2 stacks, due to the intersections of pairs of edges $((d, e)$ and (a, b) for instance). We show below the sequence of ENQUEUE and DEQUEUE operations on the queue. In conformance with the queue regimen, edges are DEQUEUEed in the order in which they were ENQUEUEed.

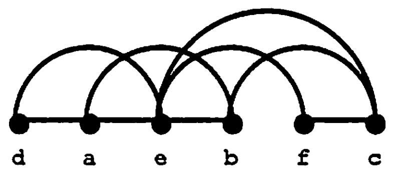

Figure 3: One-queue layout

Node	Operation	Composition of the queue
d	ENQUEUE (d, a)	(d, a)
	ENQUEUE (d, e)	$(d, a), (d, e)$
a	DEQUEUE (d, a)	(d, e)
	ENQUEUE (a, e)	$(d, e), (a, e)$
	ENQUEUE (a, b)	$(d, e), (a, e), (a, b)$
e	DEQUEUE (d, e)	$(a, e), (a, b)$
	DEQUEUE (a, e)	(a, b)
	ENQUEUE (e, b)	$(a, b), (e, b)$
	ENQUEUE (e, f)	$(a, b), (e, b), (e, f)$
	ENQUEUE (e, c)	$(a, b), (e, b), (e, f), (e, c)$
b	DEQUEUE (a, b)	$(e, b), (e, f), (e, c)$
	DEQUEUE (e, b)	$(e, f), (e, c)$
	ENQUEUE (b, c)	$(e, f), (e, c), (b, c)$
f	DEQUEUE (e, f)	$(e, c), (b, c)$
	ENQUEUE (f, c)	$(e, c), (b, c), (f, c)$
c	DEQUEUE (e, c)	$(b, c), (f, c)$
	DEQUEUE (b, c)	(f, c)
	DEQUEUE (f, c)	-

Table 2: Queue Layout

GIQ layout. We show now a single GIQ layout for a linearization of the sample graph which does not admit either a single stack layout or single queue layout. The linearization (a, b, c, d, e, f) (see Figure 4) contains a 3-twist (edges (a, b),(b, e) and (c, f)) and thus cannot be laid out using fewer than 3 stacks. It also contains a 2-rainbow (edges (a, d) and (b, c)) and thus cannot be laid out using fewer than 2 queues. To lay out the linearization with a GIQ, we just choose the place of INSERTion of a new edge so as to keep edges in the GIQ in the order in which they are to be REMOVEd. There are several ways of dealing with edges with common source[2] or destination. In the example, we chose to INSERT edges with common source in the order of their destination. This leads to the following sequence of INSERT and REMOVE operations. We give a formal description and analysis of the algorithm in the next section.

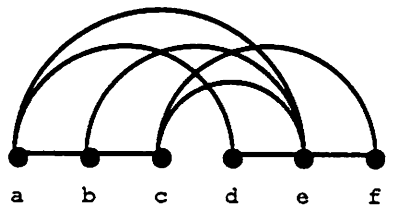

Figure 4: Linearization used for the GIQ layout

2 GIQ Layout

In this section we describe an algorithm for computing the GIQ layout of a linearized graph, i.e., the sequence of INSERT and REMOVE operations on the edges of the graph and the position of each insertion. The input to the algorithm is the adjacency list of the graph, given in the order of the linearization; the output is the position of insertion of each edge in the GIQ.

The strategy of the algorithm is to scan the linearized graph from left to right, processing the edges in the order in which they appear in the adjacency list. Edges are INSERTed in and REMOVEed from an ordered list which simulates a GIQ. The position of an edge in the list at the time of its insertion is the same as the parameter to the INSERT operation in the GIQ.

Assumptions and definitions. The nodes of the linearized graph are laid in a line from left to right, and are labeled with their ranks in the linearization: for any pair of nodes a and b, the assertion $a < b$ denotes the fact that a is to the left of b.

[2]The *source* of an edge is the left node to which it is incident in the linearization; the *destination* of an edge is the right node to which it is incident.

Node	Operation		Composition of the GIQ
a	INSERT-1	(a,b)	(a,b)
	INSERT-2	(a,d)	$(a,b),(a,d)$
	INSERT-3	(a,e)	$(a,b),(a,d),(a,e)$
b	REMOVE	(a,b)	$(a,d),(a,e)$
	INSERT-1	(b,c)	$(b,c),(a,d),(a,e)$
	INSERT-4	(b,e)	$(b,c),(a,d),(a,e),(b,e)$
c	REMOVE	(b,c)	$(a,d),(a,e),(b,e)$
	INSERT-4	(c,e)	$(a,d),(a,e),(b,e),(c,e)$
	INSERT-5	(c,f)	$(a,d),(a,e),(b,e),(c,e),(c,f)$
d	REMOVE	(a,d)	$(a,e),(b,e),(c,e),(c,f)$
	INSERT-4	(d,e)	$(a,e),(b,e),(c,e),(d,e),(c,f)$
e	REMOVE	(a,e)	$(b,e),(c,e),(d,e),(c,f)$
	REMOVE	(b,e)	$(c,e),(d,e),(c,f)$
	REMOVE	(c,e)	$(d,e),(c,f)$
	REMOVE	(d,e)	(c,f)
	INSERT-2	(e,f)	$(c,f),(e,f)$
f	REMOVE	(c,f)	(e,f)
	REMOVE	(e,f)	

Table 3: GIQ Layout

The *source* of edge e, *source*(e), is the left node incident to it, and its *destination*, *dest*(e), is the right node incident to it.

We say that edge e is *hanging* over node a either if a is the source of e, or if *source*(e) is to the left of a and *dest*(e) is to the right of a; equivalently, we say that a *lies under* edge e.

Edge e *crosses* edge f if *source*(e) lies under f and *dest*(f) lies under e: $source(f) \leq source(e) < dest(f) < dest(e)$. The *crossing number* of an edge in the linearization is the number of edges it crosses.

For the purpose of the algorithm, nodes of the linearized graph are split into *subnodes* corresponding to the incident edges. In the following sections, *source* and *destination* refer to the nodes incident to an edge, *left* and *right* to the subnodes of the split graph. Consider edge (c,e) in Figure 4: it has source c and destination e; after the splitting of the nodes, node c has been split into subnodes 7 to 9, and node e has been split into subnodes 12 to 16. Edge (c,e) now connects subnode 8 (left subnode) and subnode 14 (right subnode) and we thus label it (8,14) (see Figure 5).

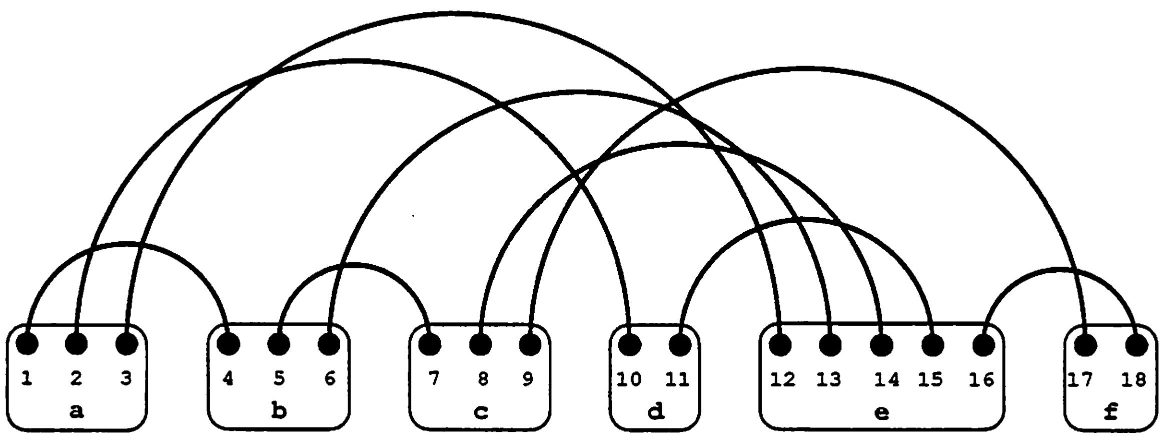

Figure 5: The example graph after splitting

Data structures. The linearized graph is represented by a variant of its adjacency list in which the nodes are listed in left-to-right order, and, for each node, the neighbors are also listed from left to right.

As a preliminary step, the algorithm organizes all the relevant information about the edges of the graph in an array called **subnodes**. Each cell of the array corresponds to the extremity of one edge (a *subnode* of the split graph), and contains the following information: *rank* of the subnode, whether it is the *source* or the *destination* of the edge, rank of the neighboring subnode and *label* of the edge (the pair *source*, *destination*).

During the scanning of the graph, we keep the labels of the edges hanging over the current node in a sorted list. Operations on this list are *insertion* (which adds a new label to the list and returns the rank of insertion edge) and *deletion* (which deletes an edge from the list). We call this list **hanging**. Since insertions and deletions in the **hanging** list account for most of the operations of the algorithm, the choice of data structure for the **hanging** list has some bearing on the total running time of the algorithm. We implement the list as a balanced order-statistic tree, specifically, the augmented red-black tree described in [6].

Finally, the crossing number of each edge is stored in an array called **crossings** ordered according to the source of the edges: the edge with leftmost source is first.

The Layout Algorithm. The algorithm that we present here is called CROSSING-NUMBER. Given a linearization of a graph, it computes the crossing number of each edge, which is easily seen to be the insertion position in the GIQ layout. CROSSING-NUMBER calls the procedures SPLIT-GRAPH and LABEL-EDGES, which set up the **subnodes** array, and COMPUTE-CROSSINGS, which actually performs the computation. We now describe the three procedures in detail, using the example of the previous section as an illustration.

Node splitting. Each node of the graph is split into as many subnodes as there are edges incident to it. We keep the subnodes ordered in an array of size twice the number of edges in the graph (18 in the example). Each subnode corresponds to one extremity of an edge (and, therefore, to one entry in the adjacency list). SPLIT-GRAPH fills each cell of the **subnodes** array with the following information: rank of the subnode, source, destination and direction of the corresponding edge; it also initializes the **left** look-up table to the leftmost subnode of each node (see Tables 4 and 5).

```
SPLIT-GRAPH :
$s \leftarrow 1$
for $node \leftarrow$ 1 to number-of-nodes($\mathcal{G}$) do
    left[$node$] $\leftarrow s$
    for $edge \leftarrow$ 1 to node-degree($node$) do
        subnodes[$s$].rank $\leftarrow s$
        get $neighbor$ from adjacency list
        if $neighbor > node$
            subnodes[$s$].direction $\leftarrow$ right
            subnodes[$s$].source $\leftarrow$ $node$
            subnodes[$s$].destination $\leftarrow$ $neighbor$
        else
            subnodes[$s$].direction $\leftarrow$ left
            subnodes[$s$].source $\leftarrow$ $neighbor$
            subnodes[$s$].destination $\leftarrow$ $node$
    $s$++
```

Edge labeling. LABEL-EDGES scans the **subnodes** array from left to right, determining the rank of the left and right subnode of each edge (the pair *left-subnode,right-subnode*). Each time it encounters a left subnode, it determines the right subnode of the corresponding edge by looking up the element of the **left** array corresponding to the destination node; it then increments this element. The label of the edge, consisting of the pair (left subnode index, right subnode index), is written in the appropriate field of the cells of the **subnodes** array corresponding to the left and right subnodes.

```
LABEL-EDGES :
for $s \leftarrow 1$ to $2 \times$ number-of-edges($\mathcal{G}$) do
    if subnodes[$s$].direction = right
        $l \leftarrow s$
        $r \leftarrow$left[subnodes[$s$].destination]
```

```
left[subnodes[$s$].destination]++
subnodes[l].label ← $(l, r)$
subnodes[r].label ← $(l, r)$
```

Node	Adjacent Nodes
a	b, d, e
b	a, c, e
c	b, e, f
d	a, e
e	a, b, c, d, f
f	c, e

Table 4: Adjacency list of the chosen linearization

Subnode	Source	Dest.	Dir.	Label
1	a	b	left	(1,4)
2	a	d	left	(2,10)
3	a	e	left	(3,12)
4	a	b	right	(1,4)
5	b	c	left	(5,7)
6	b	e	left	(6,13)
7	b	c	right	(5,7)
8	c	e	left	(8,14)
9	c	f	left	(9,17)
10	a	d	right	(2,10)
11	d	e	left	(11,15)
12	a	e	right	(3,12)
13	b	e	right	(6,13
14	c	e	right	(8,14)
15	d	e	right	(11,15)
16	d	f	left	(16,18)
17	c	f	right	(9,17)
18	e	f	right	(16,18)

Table 5: Contents of the **subnodes** array after LABEL-EDGES

Determining the crossing numbers. COMPUTE-CROSSINGS scans the subnodes of the split graph (i.e., the **subnodes** array) from left to right, inserting or removing (as appropriate) the label of the corresponding edge into or from the **hanging** list. Each time it encounters a left subnode, it inserts the incident edge into the list, and returns the position of insertion. Since the list is kept ordered by destination, the position of insertion is equal to the crossing number of the edge.

```
COMPUTE-CROSSINGS :
$c \leftarrow 1$
for $s \leftarrow 1$ to $2 \times$ number-of-edges($\mathcal{G}$) do
    if subnodes[$s$].direction = right
        INSERT(subnodes[$s$].label,*position*)
        crossings[$c$] $\leftarrow$ *position*
        $c$++
```

Correctness of the Algorithm.

Theorem 1 *Algorithm* CROSSING-NUMBER *correctly computes the crossing number of each edge in a linearized graph.*

Proof. The first two procedures of the algorithm, SPLIT-GRAPH and LABEL-EDGES, arrange and label the edges of the graph in the order of the linearization, provided the adjacency list is written in the correct format. The correct identification of the source and destination of each edge follows from the fact that the nodes are listed in the order of the linearization. The correct indexing of destination subnodes follows from the facts that neighbors of a given node are listed in the order of the linearization, and, therefore, that all right subnodes precede all left subnodes.

COMPUTE-CROSSINGS scans the subnodes from left to right, and the **hanging** list is kept ordered with respect to destination subnode. Consider edge (l, r) when it is inserted into the list. Since the edge is inserted, l is the rank of the subnode being scanned. Consider edge (l', r') not present in the list. Either it has not been inserted ($l < l'$), or it has already been removed from the list ($r' < l$), in both cases, edge (l, r) cannot cross it. Consider edge (l', r') following edge (l, r) in the list. Since the list is sorted in order of destination, it follows that $r' > r$; therefore, edge (l, r) does not cross edge (l', r'). Finally, consider edge (l', r') preceding edge (l, r) in the list. Clearly, $l > l'$, since l' is already in the list, and $r' < r$ since the list is sorted. Therefore, edge (l', r') precedes edge (l, r) in the list when (l, r) is inserted if and only if $l' < l < r' < r$. This means that all the edges crossed by edge (l, r) precede it in the list, and, therefore, the position of insertion of edge (l, r) is equal to its crossing number.□

Running time of the Algorithm.

Theorem 2 *Algorithm* CROSSING-NUMBER *computes the crossing number of each edge in a linearized graph in time $O(|E| \log c)$, where $|E|$ is the number of edges in the graph, and c is the cutwidth of the graph.*

Proof. Procedure SPLIT-GRAPH scans the adjacency list and initializes one cell of the **subnodes** array per entry. Since there are two entries per edge in the adjacency list, this clearly takes $O(|E|)$. The same reasoning applies to procedure LABEL-EDGES which initializes the remaining fields of the cells of the same array.

The loop in COMPUTE-CROSSINGS executes $2|E|$ times, and, since each edge is inserted and removed once from the list, COMPUTE-CROSSINGS consists of $|E|$ insertions and $|E|$ deletions. Since we implement the **hanging** list as a balanced order-statistic tree in which each insertion and deletion takes time $O(\log n)$, where n is the size of the tree, and since the number of edges in the list is at most equal to the cutwidth of the graph, COMPUTE-CROSSINGS runs in time $O(|E| \log c)$.□

3 Implementation Issues

3.1 Switch Graphs

The introduction of the GIQ structure in the new version of the DIOGENES methodology was motivated by "engineering" considerations: while looking at the detailed implementation of the switches required for stack and queue layout, we found that a relatively small modification of the basic switches allowed the flexibility of the GIQ regimen.

To describe the connections established by individual switches in a GIQ network, we introduce a graph representation which we call *switch graphs*. Basically, nodes of a switch graph correspond to the terminals of a hardware switch, i.e., its connections to the bundle of wires and to a processor port; edges of a switch graph represent the connections established by the switch. We therefore talk of the *number of lines*, the *direction*, and the *setting* of a switch graph.

Besides offering a convenient representation of the operations on a bounded size GIQ, switch graphs reflect the structure of the necessary switches closely enough that one can use them to model the hardware switch design which is part of the physical design of a GIQ-DIOGENES layout of a processor array. We can, therefore, use the switch graphs to estimate the cost of a proposed design.

An *n-line switch graph* $G_n(D, S)$ has $2n+1$ nodes: n *left* nodes $\{N_{left,1}, \ldots, N_{left,n}\}$, n *right* nodes $\{N_{right,1}, \ldots, N_{right,n}\}$ and a single node N_P (see Figure 6). The edges of the graph are described by the *direction* $D \in \{left, right\}$ and the *setting* $S \in \{0, 1, \ldots, n\}$.

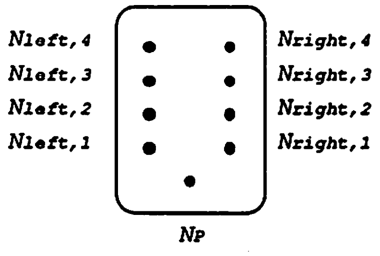

Figure 6: Nodes of a 4-line switch graph

If $S = 0$, node N_P is not connected to the rest of the switch. There are then n edges in the graph, $\langle E_1, \ldots, E_n \rangle$, each edge E_k connecting nodes $N_{left,k}$ and $N_{right,k}$ (see Figure 7).

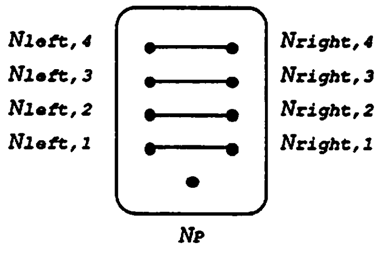

Figure 7: A 4-line switch graph with setting $S = 0$

If $S > 0$, then there is an edge E_0 connecting node N_P to node $N_{D,S}$. The rest of the edges are as follows, for $0 < i < S$, there is an edge E_i connecting nodes $N_{left,i}$ and $N_{right,i}$, and for $S \leq j < n$, there is an edge E_j connecting nodes $N_{D',j}$ and $N_{D,j+1}$ (where D' is the direction opposite to D) (see Figures 8 and 9).

A switch graph can be used to represent a switch connecting a port of a PE to a GIQ-DIOGENES network: the nodes of the switch graph correspond to wires passing through the switch, and the edges of the switch graph describe the routing established by the switch. Specifically, node N_P represents the connection to the wire coming into or departing from the port of the PE, while nodes $\{N_{left,1}, \ldots, N_{left,n}\}$ and $\{N_{right,1}, \ldots, N_{right,n}\}$ represent, respectively, the connections to wires to the left and the right of the switch. The *direction* of the switch graph indicates whether the wire connected to the port is INSERTed into the bundle or REMOVEd from it. Since we view the logical line of PEs as oriented from left to right, a n-line switch graph $G_n(right, k)$ represents a switch INSERTing a new wire into position k of a bundle of n wires, and a n-line switch graph

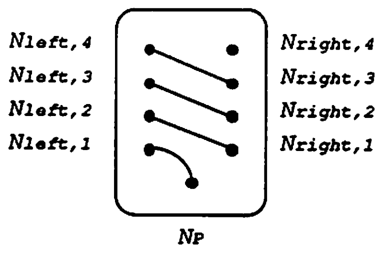

Figure 8: A 4-line left switch graph with setting $S = 1$

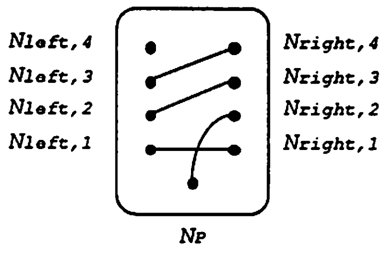

Figure 9: A 4-line right switch graph with setting $S = 2$

$G_n(left,1)$ represents a switch REMOVing the first wire from a bundle of n wires; finally, a switch graph with setting 0 represents the bypassing of a port.

To represent a whole GIQ-DIOGENES network, we use a graph obtained by interconnecting a collection of switch graphs, one for each port of every PE of the array. A network requiring a bundle of n wires and K switches can be represented by a graph composed of K interconnected n-line switch graphs. The nodes of this graph are: $\left\{N_{D,S}^{(i)} \,|\, i \in \{1,\ldots,K\}, D \in \{\mathit{left},\mathit{right}\}, S \in \{0,1,\ldots,n\}\right\} \cup \left\{N_P^{(i)} \,|\, i \in \{1,\ldots,K\}\right\}$. The edges of the graph are the edges of the switch graphs, plus a set of edges connecting each switch graph to its neighbors: each right node of each switch graph is connected to the corresponding left node of the neighboring switch graph (see Figure 10).

The connections of port node N_P^k $(1 \leq k \leq K)$ to the network are described by the direction D_k and setting S_k of switch graph k. The direction and setting of each switch graph, which are used in the physical layout of the GIQ network, can be easily derived from the result of the algorithm CROSS-OVER. One switch graph corresponds to each element of the **subnodes** array, the direction of the switch graph being the direction of the corresponding edge. The right switch graphs, corresponding to INSERT operations, have settings one greater than the crossing number of the corresponding edge, as computed by algorithm CROSSING NUMBER. The left switch graphs, corresponding to REMOVE operations all have setting 1. This asymmetry between left and right switch graphs

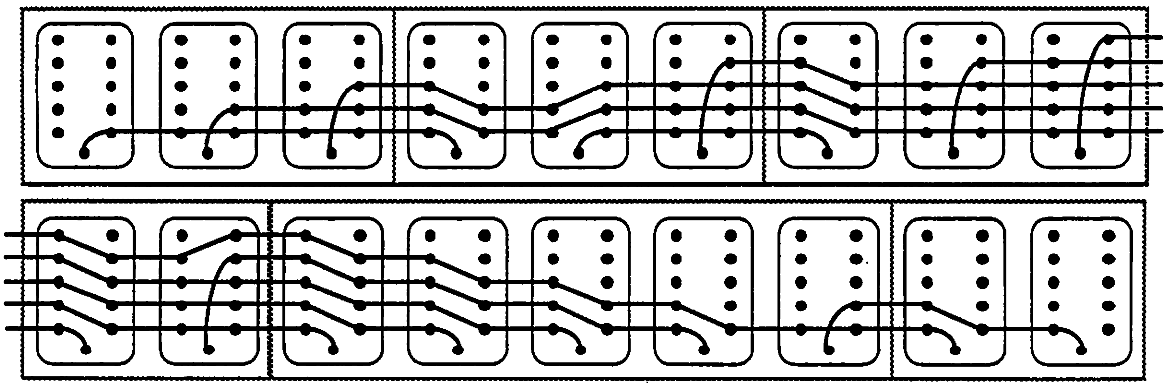

Figure 10: Switch graph representation of the example GIQ layout. The bottom row of switches is a continuation of the top row.

substantially reduces the hardware needed to implement GIQ-DIOGENES networks, as we show in the next section.

Example layout. Figure 10 represents the layout of the example graph of the previous section using GIQ switches. The graph contains nine edges, and the cutwidth of the chosen linearization (a, b, c, d, e, f) is five (between nodes c and d and d and e); the GIQ layout therefore requires nine 5-line left switch graphs with setting 1 and nine 5-line right switch graphs, whose setting is derived from the crossing number of the corresponding edge. Switch graph settings and direction for the example graph are given in Table 6.

3.2 Implementing Switch Graphs in Hardware

We turn now to the physical implementation of switch graphs. In CMOS technology, the bidirectional connections established by the switches of a DIOGENES network can be realized in a straightforward fashion using 2-to-1 multiplexers composed of two coupled complementary switches (see Figure 11). A 2-to-1 multiplexer has two "input" lines A and B, one "output" line and one control line (actually two lines, one for the control and one for its complement). Depending on the value of the control line, the output is connected to line A or B. The connection established by such a multiplexer is bidirectional.

Left switch graph. In a GIQ network, left switches implement the REMOVE operation: each wire coming from the right of the switch can either be connected to the corresponding wire to the left of the switch (if the port is by-passed), or to the next lower wire if the port is connected. This switching mechanism is realized in a straightforward fashion, using one 2-to-1 multiplexer per wire (see Figure 12). Since the multiplexers in a switch

Rank	Operation	Direction	Setting
1	INSERT-1	*right*	1
2	INSERT-2	*right*	2
3	INSERT-3	*right*	3
4	REMOVE	*left*	1
5	INSERT-1	*right*	1
6	INSERT-4	*right*	4
7	REMOVE	*left*	1
8	INSERT-4	*right*	4
9	INSERT-5	*right*	5
10	REMOVE	*left*	1
11	INSERT-4	*right*	4
12	REMOVE	*left*	1
13	REMOVE	*left*	1
14	REMOVE	*left*	1
15	REMOVE	*left*	1
16	INSERT-2	*right*	2
17	REMOVE	*left*	1
18	REMOVE	*left*	1

Table 6: Switch Setting and Direction for the Example Layout

all function in unison, one control line is sufficient to configure a switch, independently of the number of wires.

Right switch graph. The connections established by the right switch graph (INSERT operation) are more complex: each wire going out to the left of the switch can be connected to:

- the corresponding wire coming from the right (*straight* connection)
- the next lower wire coming from the right (*slanted* connection)
- the port (*port* connection)

Two 2-to-1 multiplexers are thus required for each but the lowest wire. In our design (see Figure 13), the right multiplexer establishes whether a connection is *straight* (control bit 0) or not. The left multiplexer, whose output is one of the inputs of the right multiplexer, determines whether the connection is a *port* connection (control bit 0) or a

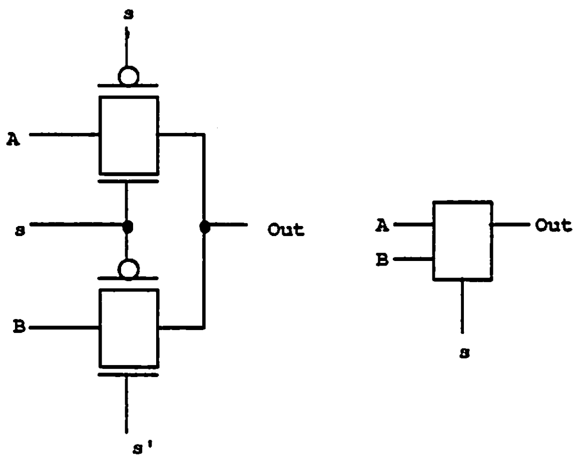

Figure 11: A CMOS multiplexer

slanted connection. An n-line switch realizing a INSERT-k operation has $k-1$ *straight* connections separated from $n-k-1$ *slanted* connections by a single *port* connection. It is thus possible to limit the number of configuration bits to one per wire; the k^{th} configuration bit controls the rightmost multiplexer of wire k and the leftmost multiplexer of wire $k+1$.

4 A Cost Comparison

In this section, we substantiate our claim that, although GIQ networks are slightly more expensive to implement than stack or queue networks of the same width, they are more economical from a global, system point of view, due to their flexibility. In other terms, a GIQ network is considerably more powerful than a stack or queue network of the same width.

4.1 Hardware Cost

The cost of switches in a DIOGENES network, whether it is implemented using stacks, queues or GIQs, can be partitioned between switching cost (the complexity of the switches themselves) and configuration cost (the cost of storing the configuration of each switch). We base the hardware cost comparison of the different techniques on our proposed implementation of CMOS switch graph. Moreover, we do not consider the details of the configuration cost, since it may vary greatly depending on the purpose of the network

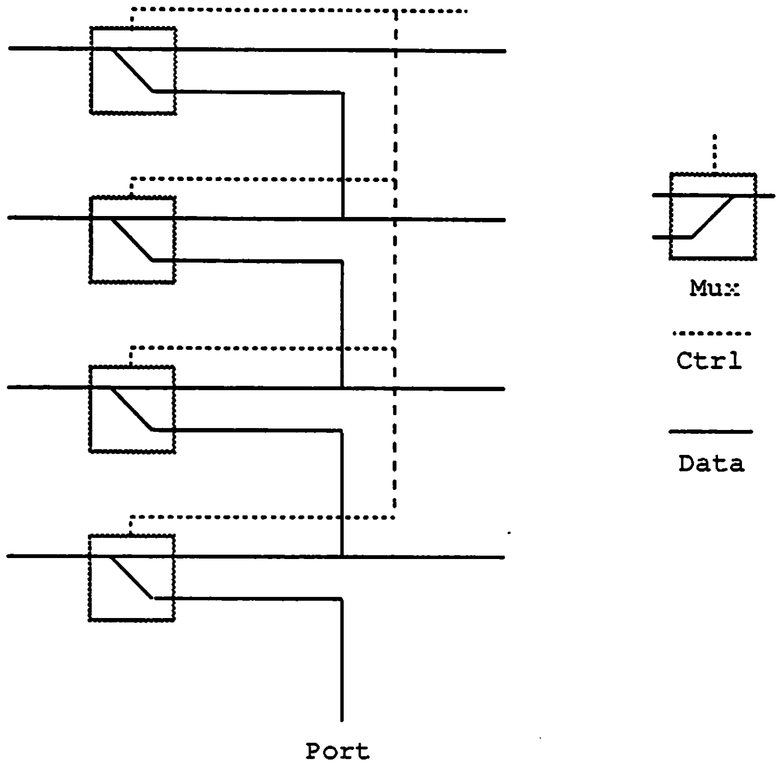

Figure 12: A 4-line Left switch

(fault-tolerance or dynamic reconfigurability). In the following analysis, all computations are for 1-bit wide data paths; they scale linearly to wider paths.

POP, DEQUEUE, and REMOVE operations. The POP, DEQUEUE, and REMOVE operations can all be represented by the same switch graph, i.e., a left switch graph with setting 1; they can, thus, be implemented at the same cost, both for switching and configuration. One 2-to-1 multiplexer is required per wire in the bundle traversing the switch, and one bit is needed to store the configuration of the switch (active or not).

PUSH, ENQUEUE and INSERT operations. The PUSH operation on a stack is the exact reverse of a POP operation and can, therefore, be represented by a right switch graph with setting 1. The hardware required for the implementation of the switch is the same as for the POP operation, i.e., one 2-to-1 multiplexer per wire and one control bit per switch.

To perform an ENQUEUE operation, it is necessary to keep track of the population of the queue at each node, namely, the number of edges hanging over the node. The

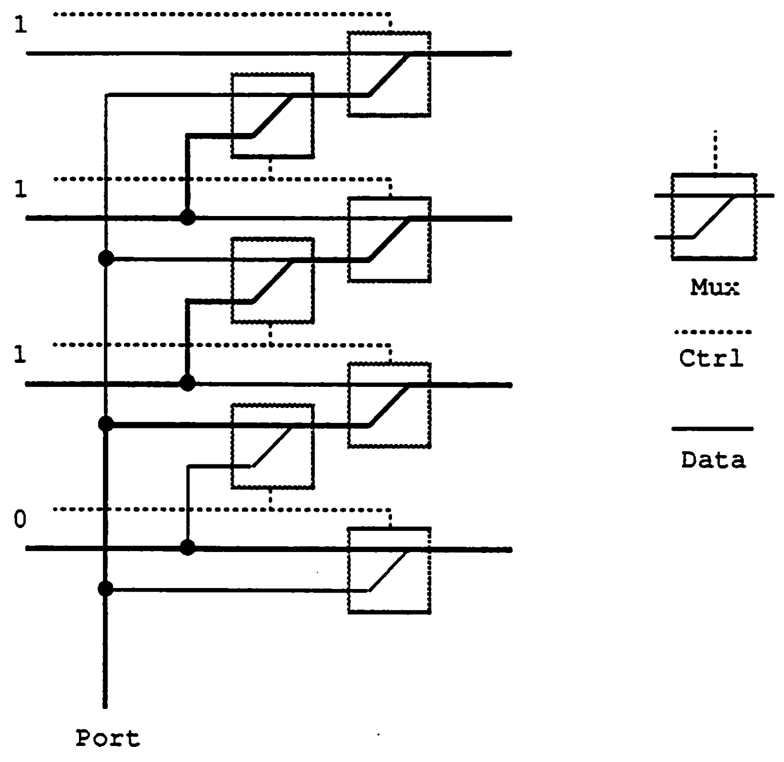

Figure 13: Right switch

connections established by a switch performing an ENQUEUE operation on a queue with population p are as follows: wire p going to the right is connected to the port, all other wires going to the right are connected to the corresponding wires coming from the left. One 2-to-1 multiplexer is thus required per wire. Additionally, one control bit is required per wire.

As was shown in the previous section, two 2-to-1 multiplexers per line are required to implement the INSERT operation; therefore, the hardware cost of the switching parts of a GIQ-based DIOGENES network is roughly fifty percent higher than the cost of a stack-based or queue-based network *of same width.* The configuration cost is the same as for a queue-based network.

In the next section, we see that the system-level flexibility of the GIQ regimen gives it a considerable advantage from a system point of view.

4.2 System Cost

Stack-based and queue-based layouts each have their particular strengths and weaknesses: some families of graphs which can be efficiently laid out with one strategy are often a problem for the other. This means that neither of the strategies can efficiently realize *all* families of graphs. Since a GIQ can simulate both a stack and a queue, it guarantees an efficient layout for all graphs that admit either an efficient stack-based layout or an efficient queue-based layout (one can simply use a linearization derived from the stack-based or queue-based layout). In fact, even when there is no stack-based or queue-based layout, the designer can use *any* optimization technique to derive an efficient layout, knowing that the resulting linearization can be implemented using a single bundle of wires following the GIQ regimen. In this section, we present examples of specific graphs for which the GIQ-based layout leads to significant cost reduction over either stack-based or queue-based layouts; we also illustrate how the flexibility of the regimen allows the designer to address some other issues, such as regularity of structure and usage of wires.

Ternary hypercube. Some families of graphs do not lend themselves to efficient stack-based layouts. In particular, the ternary hypercube $TC(n)$, which has $N = 3^n$ nodes, can be laid out using a logarithmic number of queues, namely $2n$, but requires exponentially more stacks, namely $\Omega(N^{1/9-\epsilon})$ [8].

Binary tree. In contrast, there are cases in which a stack-based layout is more "efficient" than a queue-based layout: a depth-d complete binary tree, with $n = 2^d$ nodes, can be laid out using a single queue, but any such layout has cutwidth at least $(2n - 2)/(d+1) = \Omega(n/\log n)$ [9]. The same binary tree can be laid out using a single stack with only $O(d)$ cutwidth.

Trade-offs between stacknumber and stackwidth. In a multiple stack (multiple queue) layout, each stack (queue) requires its own bundle of wires, complete with control mechanism. Once the number of stacks (queues) has been decided upon, the freedom of the implementor is severely restricted.

Building on the base cases described in [5], [14] exposes a family of families of graphs $\{\{G_{k,n}\}_{n\in\mathcal{N}}\}_{k\in\mathcal{N}}$, (where $\mathcal{N}$ denotes the positive integers); these graphs have the following property. For all k,

- any k-stack layout of a graph in the family $\{\{G_{k,n}\}_{n\in N}\}$ must have cutwidth proportional to the size of the graph;
- every graph in the family admits a $(k+1)$-stack layout of cutwidth $O(k)$.

Reusability of wires. The last consequence of the ability of a GIQ to realize any linearization of any graph is a better overall usage of the wires. In a multiple stack (queue) DIOGENES array, wires cannot be shared across bundles, so one must *a priori* partition the available wires among the bundles. In contrast, a GIQ-DIOGENES array has only one bundle, so the cutwidth can be allocated in a way that best serves the layout of the particular array of interest. One can see the benefits of this allocability in the layout of an $m \times n$ mesh.

One can realize a two-stack layout of a mesh by laying out the nodes in a snake-like fashion along the smaller dimension (see Figure 14). This layout has cutwidth $2\min(m,n)$, but only half of the lines are used at any given time: whenever one stack is full, the other one is empty.

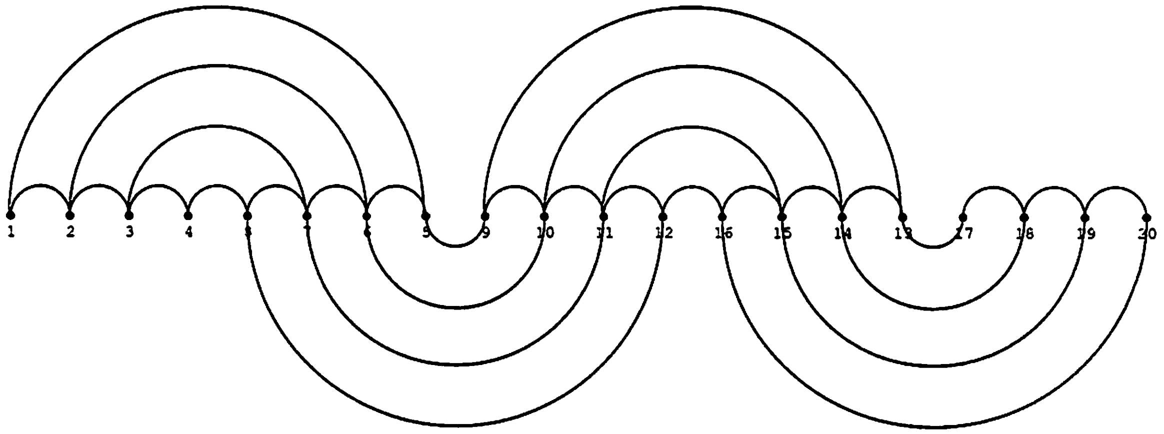

Figure 14: 2-stack layout of a 4×5 mesh

Since we can realize any linearization with a single GIQ, we can choose the ordering of the graph according to any criterion we which to optimize. For instance, one could want to maintain some "structural" properties of the original graph. In the case of the mesh, for example, it could be desirable to insure that edges of the same dimension all have the same length. In the row major layout of a 4×5 mesh (see Figure 15) all row edges have length 1 and all column edges have length 4. For an arbitrary graph, one could choose to minimize bandwidth, cutwidth or any other criterion, use a heuristic to obtain an acceptable linearization, and implement the layout using a GIQ network.

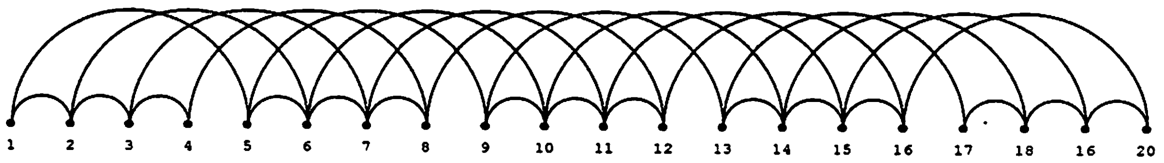

Figure 15: GIQ layout of a 4×5 mesh

5 Conclusion

We have shown in this paper how a relatively simple and inexpensive modification of the switches used in a DIOGENES network can give a much greater flexibility in the layout regimen. The GIQ regimen, which we introduced here, allows much simpler layout of fault-tolerant networks. The advantages reside both in a simpler layout algorithm and a lower overall cost. Another consequence of the ability to lay out any graph (with limited degree and cutwidth), is that, if we compute the settings for a selection of graphs off-line, it is possible to switch configuration during the computation.

As a proof of concept, we are currently designing a 32-processor array in which each processor is connected to the network by six ports; there are eight datalines in the bundle. In our design, the network can adopt any of a set of precomputed configuration. The reconfiguration of the network simply consists in loading another set of settings for the switches. This circuit will be able to assume such different topologies as butterfly, hypercube, three-dimensional mesh and de Bruijn graph. Furthermore, we are investigating strategies that time-multiplex the wires and the ports pf a GIQ-DIOGENES array, to vercome the physical limitations of fixed cutwidth and node-degree.

References

[1] K.P. Belkhale and P. Banerjee (1991): Reconfiguration strategies for VLSI processor arrays and trees using a modified DIOGENES approach. *IEEE Trans. Comp.*, to appear.

[2] F. Bernhart and P.C. Kainen (1979): The book thickness of a graph. *J. Comb. Th. (B) 27*, 320-331.

[3] J.F. Buss, A.L. Rosenberg, J.D. Knott (1989): Vertex-types in book-embeddings. *SIAM J. Discrete Math. 2*, 156-175.

[4] F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1983): DIOGENES – A methodology for designing fault-tolerant processor arrays. *13th Intl. Conf. on Fault-Tolerant Computing*, 26-32.

[5] F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1987): Embedding graphs in books: A layout problem with applications to VLSI design. *SIAM J. Algebr. Discr. Meth. 8*, 33-58.

[6] T.H.Cormen, C.E. Leiserson, R.L. Rivest (1990): Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts, 281-285.

[7] E. Gurari and I.H. Sudborough (1984): Improved dynamic programming algorithms for bandwidth minimization and the min-cut linear arrangement problem. *J. Algorithms 5*, 531-546.

[8] L.S. Heath, F.T. Leighton, A.L. Rosenberg (1991): Comparing queues and stacks as mechanisms for laying out graphs. *SIAM J. Discrete Math.*, to appear. See also, Graph layouts using queues. *28th Allerton Conf. on Communication, Computation, and Control*, 305-314.

[9] L.S. Heath and A.L. Rosenberg (1992): Laying out graphs using queues. *SIAM J. Comput.*, to appear.

[10] L.S. Heath, A.L. Rosenberg, B.T. Smith (1988): The physical mapping problem for parallel architectures. *J. ACM 35*, 603-634.

[11] J.D. Knott and A.L. Rosenberg (1987): The DIOGENES design methodology: On minimizing maximum wire-run. *2nd Intl. Conf. on Supercomputing*, 260-268.

[12] A.L. Rosenberg (1983): The Diogenes approach to testable fault-tolerant arrays of processors. *IEEE Trans. Comp., C-32*, 902-910.

[13] A.L. Rosenberg (1984): On designing fault-tolerant VSLI processor arrays. In *Advances in Computing Research 2* (F.P. Preparata, ed.) JAI Press, Greenwich, CT, pp. 181-204.

[14] E. Stöhr (1988): A trade-off between page number and page width of book embeddings of graphs. *Inform. Comput. 79*, 155-162.

[15] N. Weste, K. Eshraghian (1988): Principles of CMOS VLSI Design. Addison-Wesley, Reading, Massachusetts.